EXPRESS MAIL NO. EV347012631US


# PCT

WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

| | | |
|---|---|---|
| (51) International Patent Classification 7 : G06F 9/38 | A1 | (11) International Publication Number: WO 00/33186<br>(43) International Publication Date: 8 June 2000 (08.06.00) |

(21) International Application Number: PCT/US99/28822

(22) International Filing Date: 3 December 1999 (03.12.99)

(30) Priority Data:
09/204,536 3 December 1998 (03.12.98) US



(71) Applicant: SUN MICROSYSTEMS, INC. [US/US]; 901 San Antonio Road, M/S PAL01-521, Palo Alto, CA 94303 (US).

(72) Inventor: TREMBLAY, Marc; 140 Hanna Way, Menlo Park, CA 94025 (US).

(74) Agents: KOESTNER, Ken, J. et al.; Skjerven, Morrill, MacPherson, Franklin & Friel LLP, Suite 700, 25 Metro Drive, San Jose, CA 95110 (US).



(81) Designated States: JP, KR, European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

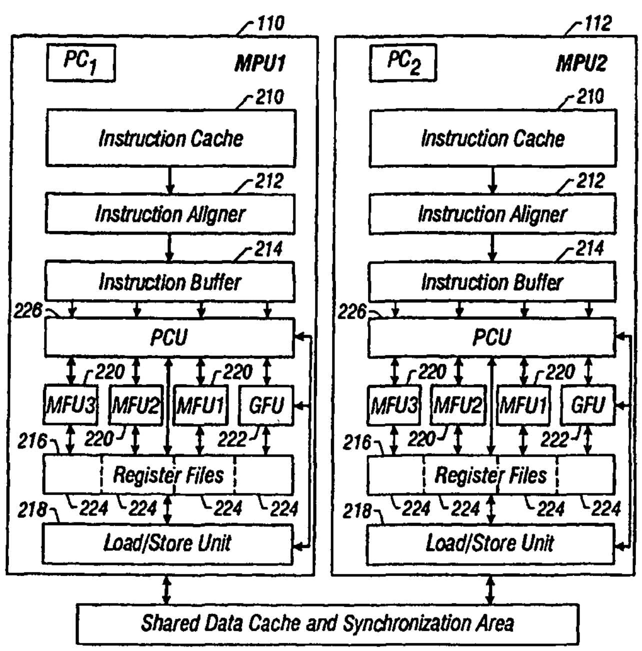
**Published**

*With international search report.*

*Before the expiration of the time limit for amending the claims and to be republished in the event of the receipt of amendments.*


(54) Title: VARIABLE ISSUE-WIDTH VLIW PROCESSOR

(57) Abstract





A processor has a flexible architecture that efficiently handles computing applications having a range of instruction-level parallelism from a very low degree to a very high degree of instruction-level parallelism. The processor includes a plurality of processing units, an individual processing unit of the plurality of processing units including a multiple-instruction parallel execution path. For computing applications having a low degree of instruction-level parallelism, the processor includes control logic that controls the plurality of processing units to execute instructions mutually independently in a plurality of independent execution threads. For computing applications having a high degree of instruction-level parallelism, the processor further includes control logic that controls the plurality of processing units with a low thread synchronization to operate in combination using spatial software pipelining in the manner of a single wide-issue processor. The control logic in the processor alternatively controls the plurality of processing units to operate: (1) in a multiple-thread operation on the basis of a highly parallel structure including multiple independent parallel execution paths for executing in parallel across threads and a multiple-instruction parallel pathway within a thread, and (2) in a single-thread wide-issue operation on the basis of the highly parallel structure including multiple parallel execution paths with low level synchronization for executing the single wide-issue thread. The multiple independent parallel execution paths include functional units that execute an instruction set including special data-handling instructions that are advantageous in a multiple-thread environment.

# VARIABLE ISSUE-WIDTH VLIW PROCESSOR

## TECHNICAL FIELD

The present invention relates to processors. More specifically, the present invention relates to architectures for Very Long Instruction Word (VLIW) processors.

## BACKGROUND ART

One technique for improving the performance of processors is parallel execution of multiple instructions to allow the instruction execution rate to exceed the clock rate. Various types of parallel processors have been developed including Very Long Instruction Word (VLIW) processors that use multiple, independent functional units to execute multiple instructions in parallel. VLIW processors package multiple operations into one very long instruction, the multiple operations being determined by sub-instructions that are applied to the independent functional units. An instruction has a set of fields corresponding to each functional unit. Typical bit lengths of a subinstruction commonly range from 16 to 24 bits per functional unit to produce an instruction length often in a range from 112 to 168 bits.

The multiple functional units are kept busy by maintaining a code sequence with sufficient operations to keep instructions scheduled. A VLIW processor often uses a technique called trace scheduling to maintain scheduling efficiency by unrolling loops and scheduling code across basic function blocks. Trace scheduling also improves efficiency by allowing instructions to move across branch points.

Limitations of VLIW processing include limited parallelism, limited hardware resources, and a vast increase in code size. A limited amount of parallelism is available in instruction sequences. Unless loops are unrolled a very large number of times, insufficient operations are available to fill the instruction capacity of the functional units. The operational capacity of a VLIW processor is not determined by the number of functional units alone. The capacity also depends on the depth of the operational pipeline of the operational units. Several operational units such as the memory, branching controller, and floating point functional units, are pipelined and perform a much larger number of operations than can be executed in parallel. For example, a floating point pipeline with a depth of eight steps has two operations issued on a clock cycle that cannot depend on any of the operations already within the floating point pipeline. Accordingly, the actual number of independent operations is approximately equal to the average pipeline depth times the number of execution units. Consequently, the number of operations needed to maintain a maximum efficiency of operation for a VLIW processor with four functional units is twelve to sixteen.

Limited hardware resources are a problem, not only because of duplication of functional units but more importantly due to a large increase in memory and register file bandwidth. A large number of read and write ports are necessary for accessing the register file, imposing a bandwidth that is difficult to support without a large

cost in the size of the register file and degradation in clock speed. As the number of ports increases, the complexity of the memory system further increases. To allow multiple memory accesses in parallel, the memory is divided into multiple banks having different addresses to reduce the likelihood that multiple operations in a single instruction have conflicting accesses that cause the processor to stall since synchrony must be maintained between the functional units.

Code size is a problem for several reasons. The generation of sufficient operations in a nonbranching code fragment requires substantial unrolling of loops, increasing the code size. Also, instructions that are not full include unused subinstructions that waste code space, increasing code size. Furthermore, the increase in the size of storages such as the register file increase the number of bits in the instruction for addressing registers in the register file.

A challenge in the design of VLIW processors is effective exploitation of instruction-level parallelism. Highly parallel computing applications that have few data dependencies and few branches are executed most efficiently using a wide VLIW processor with a greater number of subinstructions in a VLIW group. However many computing applications are not highly parallel and include branches or data dependencies that waste space in instruction memory and cause stalling. Referring to **FIGURE 1A**, a graph illustrates a comparison of instruction issue efficiency and processor size for typical computing applications having generally average instruction-level parallelism as VLIW group width is varied. The left axis of the graph relates to an instruction-level parallelism plot **10** that depicts the number of instructions executed per cycle against VLIW issue width. The right axis of the graph relates to a relative processor size plot **12** that shows relative processor size in relation to VLIW issue width.

The graphs shown in **FIGURE 1A** illustrate that larger VLIW issue widths disadvantageously achieve little improvement in execution efficiency at a great cost in circuit area for a typical range of computing applications having an average level of instruction-level parallelism. However, some computing applications, for example graphics and video applications, inherently have a high level of instruction-level parallelism. Referring to **FIGURE 1B**, a second set of graphs shows the efficiency characteristics of a computing application that has a high degree of instruction-level parallelism. Such a computing application may operate on multiple channels of graphic data where the data in multiple channels are mutually independent so that no data dependencies arise. In applications having an inherent high instruction-level parallelism, a wide VLIW issue size is justified and results in highly efficient performance.

What is needed is a processor architecture that supports efficient handling of computing applications having a wide range of instruction-level parallelism.

## DISCLOSURE OF INVENTION

A processor has a flexible architecture that efficiently handles computing applications having a range of instruction-level parallelism from a very low degree to a very high degree of instruction-level parallelism. The processor includes a plurality of processing units, an individual processing unit of the plurality of processing units including a multiple-instruction parallel execution path. For computing applications having a low degree of instruction-level parallelism, the processor includes control logic that controls the plurality of processing units to execute instructions mutually independently in a plurality of independent execution threads. For computing applications having a high degree of instruction-level parallelism, the processor further includes control logic that controls the plurality of processing units to operate in combination using spatial software pipelining in the manner of a single wide-issue processor.

The control logic in the processor alternatively controls the plurality of processing units to operate: (1) in a multiple-thread operation on the basis of a highly parallel structure including multiple independent parallel execution paths for executing in parallel across threads and a multiple-instruction parallel pathway within a thread, and (2) in a single-thread wide-issue operation on the basis of the highly parallel structure including multiple parallel execution paths for executing the single wide-issue thread. The multiple independent parallel execution paths include functional units that execute an instruction set including special data-handling instructions that are advantageous in a multiple-thread environment.

In an illustrative embodiment, a Very Long Instruction Word (VLIW) processor includes a plurality of independent processors on a single integrated circuit chip and a control logic that controls an execution environment in which multiple independent execution threads execute simultaneously. For instructions having a low level of instruction-level parallelism, the control logic operates the plurality of independent processors to execute a corresponding plurality of execution threads in parallel. The plurality of execution threads are variously controlled to operate independently or in combination. The control logic thus supports operation of the plurality of execution threads in the same application, in different applications, in the operating system, in the runtime environment (for example, garbage collection), and the like.

For applications with a high level of instruction-level parallelism, the VLIW processor executes a plurality of instructions in parallel on the plurality of independent processors using low thread synchronization overhead to operate with the same level of performance as an increased width VLIW processor. For example, in an illustrative embodiment, a VLIW processor includes two independent four-wide VLIW processors to selectively operate either as an eight-wide VLIW processor or two independent four-wide VLIW processors that mutually execute separate threads. Each of the independent processors includes a very rich set of functional units to form, in combination, a highly powerful processor, without the complexity of implementing the extensive control circuitry and connections of an eight-wide VLIW processor.

A processor has an improved architecture for applications having a wide range of parallelism through the usage of a highly parallel structure including multiple independent parallel execution paths for executing in parallel across threads and a multiple-instruction parallel pathway within a thread. The processor includes interconnections between the multiple independent parallel execution paths to selectively operate the multiple independent parallel execution paths either in coordination as a combined-width VLIW processor with subinstructions in the multiple independent parallel execution paths extending across the execution paths, or independently as a plurality of threads. The multiple independent parallel execution paths include functional units that execute an instruction set including special data-handling instructions that are advantageous in a multiple-thread environment.

In accordance with one embodiment of the present invention, a general-purpose processor includes two independent processor elements in a single integrated circuit die. The dual independent processor elements advantageously execute selectively in multiple modes including a multiple-threaded mode in which two independent threads concurrently execute during multiple-threading operation, and a single-thread mode in which a single VLIW instruction word is executed across all of the functional units in the two independent processor elements. The instructions in an instruction group execute on separate functional units.

In the independent operating mode, the two threads execute independently on the respective VLIW processor elements, each of which includes a plurality of powerful functional units that execute in parallel. In the illustrative embodiment, the VLIW processor elements have four functional units including three media functional units and one general functional unit. In the single-thread mode, the VLIW processor elements execute in combination on eight functional units including six media functional units and two general functional units. All of the illustrative media functional units include an instruction that executes both a multiply and an add in a single cycle, either floating point or fixed point.

## BRIEF DESCRIPTION OF DRAWINGS

The features of the described embodiments are specifically set forth in the appended claims. However, embodiments of the invention relating to both structure and method of operation, may best be understood by referring to the following description and accompanying drawings.

**FIGUREs 1A** and **1B** are graphs that illustrate a comparison of instruction issue efficiency and processor size as VLIW group width is varied for applications having an average level of instruction-level parallelism and for applications having a high degree of instruction-level parallelism, respectively.

**FIGURE 2** is a schematic block diagram illustrating a single integrated circuit chip implementation of a processor in accordance with an embodiment of the present invention.

**FIGURE 3** is a schematic block diagram showing the core of the processor.

**FIGURE 4** is a schematic block diagram that illustrates an embodiment of the split register file that is suitable for usage in the processor.

**FIGURES 5A** and **5B** are schematic block diagrams showing respectively (a) a logical view of the register file and functional units in the processor, and (b) a combination of two groups of register files and functional units to form an 8-wide VLIW processor.

**FIGURE 6** is a pictorial schematic diagram depicting an example of instruction execution among a plurality of media functional units.

**FIGURE 7** is a schematic timing diagram that illustrates timing of the processor pipeline.

**FIGUREs 8A-8C** are respectively, a schematic block diagram showing an embodiment of a general functional unit, a simplified schematic timing diagram showing timing of a general functional unit pipeline, and a bypass diagram showing possible bypasses for the general functional unit.

**FIGURE 9** is a simplified schematic timing diagram illustrating timing of media functional unit pipelines.

**FIGUREs 10A-10C** respectively show an instruction sequence table, and two pipeline diagrams illustrating execution of a VLIW group which shows stall operation for a five-cycle pair instruction and a seven-cycle pair instruction.

**FIGUREs 11A-11C** are pipeline diagrams illustrating synchronization of pair instructions in a group.

**FIGUREs 12A, 12B, 12C,** and **12D** are respective schematic block diagrams illustrating the pipeline control unit segments allocated to all of the functional units GFU, MFU1, MFU2, and MFU3.

**FIGURE 13** is a schematic block diagram that illustrates a load annex block within the pipeline control unit.

The use of the same reference symbols in different drawings indicates similar or identical items.

## MODES FOR CARRYING OUT THE INVENTION

Referring to **FIGURE 2**, a schematic block diagram illustrates a processor **100** having an improved architecture for multiple-thread operation on the basis of a highly parallel structure including multiple independent parallel execution paths, shown herein as two media processing units **110** and **112**. The execution paths execute in parallel across threads and include a multiple-instruction parallel pathway within a thread. The multiple independent parallel execution paths include functional units executing an instruction set having special

data-handling instructions that are advantageous in a multiple-thread environment.

The multiple-threading architecture of the processor **100** is advantageous for usage in executing multiple-threaded applications using a language such as the Java™ language running under a multiple-threaded operating system on a multiple-threaded Java Virtual Machine™. The illustrative processor **100** includes two independent processor elements, the media processing units **110** and **112**, forming two independent parallel execution paths. A language that supports multiple threads, such as the Java™ programming language generates two threads that respectively execute in the two parallel execution paths with very little overhead incurred. The special instructions executed by the multiple-threaded processor include instructions for accessing arrays, and instructions that support garbage collection. (Java™, Sun, Sun Microsystems and the Sun Logo are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. All SPARC trademarks, including UltraSPARC I and UltraSPARC II, are used under license and are trademarks of SPARC International, Inc. in the United States and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.)

A single integrated circuit chip implementation of a processor **100** includes a memory interface **102**, a geometry decompressor **104**, the two media processing units **110** and **112**, a shared data cache **106**, and several interface controllers. The interface controllers support an interactive graphics environment with real-time constraints by integrating fundamental components of memory, graphics, and input/output bridge functionality on a single die. The components are mutually linked and closely linked to the processor core with high bandwidth, low-latency communication channels to manage multiple high-bandwidth data streams efficiently and with a low response time. The interface controllers include a an UltraPort Architecture Interconnect (UPA) controller **116** and a peripheral component interconnect (PCI) controller **120**. The illustrative memory interface **102** is a direct Rambus dynamic RAM (DRDRAM) controller. The shared data cache **106** is a dual-ported storage that is shared among the media processing units **110** and **112** with one port allocated to each media processing unit. The data cache **106** is four-way set associative, follows a write-back protocol, and supports hits in the fill buffer (not shown). The data cache **106** allows fast data sharing and eliminates the need for a complex, error-prone cache coherency protocol between the media processing units **110** and **112**.

The UPA controller **116** is a custom interface that attains a suitable balance between high-performance computational and graphic subsystems. The UPA is a cache-coherent, processor-memory interconnect. The UPA attains several advantageous characteristics including a scaleable bandwidth through support of multiple bused interconnects for data and addresses, packets that are switched for improved bus utilization, higher bandwidth, and precise interrupt processing. The UPA performs low latency memory accesses with high throughput paths to memory. The UPA includes a buffered cross-bar memory interface for increased bandwidth and improved scaleability. The UPA supports high-performance graphics with two-cycle single-word writes on the 64-bit UPA interconnect. The UPA interconnect architecture utilizes point-to-point packet switched

messages from a centralized system controller to maintain cache coherence. Packet switching improves bus bandwidth utilization by removing the latencies commonly associated with transaction-based designs.

The PCI controller **120** is used as the primary system I/O interface for connecting standard, high-volume, low-cost peripheral devices, although other standard interfaces may also be used. The PCI bus effectively transfers data among high bandwidth peripherals and low bandwidth peripherals, such as CD-ROM players, DVD players, and digital cameras.

Two media processing units **110** and **112** are included in a single integrated circuit chip to support an execution environment exploiting thread level parallelism in which two independent threads can execute simultaneously. The threads may arise from any sources such as the same application, different applications, the operating system, or the runtime environment. Parallelism is exploited at the thread level since parallelism is rare beyond four, or even two, instructions per cycle in general purpose code. For example, the illustrative processor **100** is an eight-wide machine with eight execution units for executing instructions. A typical "general-purpose" processing code has an instruction level parallelism of about two so that, on average, most (about six) of the eight execution units would be idle at any time. The illustrative processor **100** employs thread level parallelism and operates on two independent threads, possibly attaining twice the performance of a processor having the same resources and clock rate but utilizing traditional non-thread parallelism.

Thread level parallelism is particularly useful for Java™ applications, which are bound to have multiple threads of execution. Java™ methods including "suspend", "resume", "sleep", and the like include effective support for threaded program code. In addition, Java™ class libraries are thread-safe to promote parallelism. Furthermore, the thread model of the processor **100** supports a dynamic compiler which runs as a separate thread using one media processing unit **110** while the second media processing unit **112** is used by the current application. In the illustrative system, the compiler applies optimizations based on "on-the-fly" profile feedback information while dynamically modifying the executing code to improve execution on each subsequent run. For example, a "garbage collector" may be executed on a first media processing unit **110**, copying objects or gathering pointer information, while the application is executing on the other media processing unit **112**.

Although the processor **100** shown in **FIGURE 2** includes two processing units on an integrated circuit chip, the architecture is highly scaleable so that one to several closely-coupled processors may be formed in a message-based coherent architecture and resident on the same die to process multiple threads of execution. Thus, in the processor **100**, a limitation on the number of processors formed on a single die thus arises from capacity constraints of integrated circuit technology rather than from architectural constraints relating to the interactions and interconnections between processors.

The processor **100** is a general-purpose processor that includes the media processing units **110** and **112**, two independent processor elements in a single integrated circuit die. The dual independent processor elements

110 and 112 advantageously execute two independent threads concurrently during multiple-threading operation. When only a single thread executes on the processor 100, one of the two processor elements executes the thread, the second processor element is advantageously used to perform garbage collection, Just-In-Time (JIT) compilation, and the like. In the illustrative processor 100, the independent processor elements 110 and 112 are Very Long Instruction Word (VLIW) processors. For example, one illustrative processor 100 includes two independent Very Long Instruction Word (VLIW) processor elements, each of which executes an instruction group or instruction packet that includes up to four instructions. Each of the instructions in an instruction group executes on a separate functional unit.

The usage of a VLIW processor advantageously reduces complexity by avoiding usage of various structures such as schedulers or reorder buffers that are used in superscalar machines to handle data dependencies. A VLIW processor typically uses software scheduling and software checking to avoid data conflicts and dependencies, greatly simplifying hardware control circuits.

The two threads execute independently on the respective VLIW processor elements 110 and 112, each of, which includes a plurality of powerful functional units that execute in parallel. In the illustrative embodiment shown in **FIGURE 3**, the VLIW processor elements 110 and 112 have four functional units including three media functional units 220 and one general functional unit 222. All of the illustrative media functional units 220 include an instruction that executes both a multiply and an add in a single cycle, either floating point or fixed point. Thus, a processor with two VLIW processor elements can execute twelve floating point operations each cycle. At a 500 MHz execution rate, for example, the processor runs at a six gigaflop rate, even without accounting for general functional unit operation.

Referring to **FIGURE 3**, a schematic block diagram shows the core of the processor 100. The media processing units 110 and 112 each include an instruction cache 210, an instruction aligner 212, an instruction buffer 214, a pipeline control unit 226, a split register file 216, a plurality of execution units, and a load/store unit 218. In the illustrative processor 100, the media processing units 110 and 112 use a plurality of execution units for executing instructions. The execution units for a media processing unit 110 include three media functional units (MFU) 220 and one general functional unit (GFU) 222.

An individual independent parallel execution path 110 or 112 has operational units including instruction supply blocks and instruction preparation blocks, functional units 220 and 222, and a register file 216 that are separate and independent from the operational units of other paths of the multiple independent parallel execution paths. The instruction supply blocks include a separate instruction cache 210 for the individual independent parallel execution paths, however the multiple independent parallel execution paths share a single data cache 106 since multiple threads sometimes share data. The data cache 106 is dual-ported, allowing data access in both execution paths 110 and 112 in a single cycle. Sharing of the data cache 106 among independent processor elements 110 and 112 advantageously simplifies data handling, avoiding a need for a cache coordination protocol

and the overhead incurred in controlling the protocol.

In addition to the instruction cache **210**, the instruction supply blocks in an execution path include the instruction aligner **212**, and the instruction buffer **214** that precisely format and align a full instruction group of four instructions to prepare to access the register file **216**. An individual execution path has a single register file **216** that is physically split into multiple register file segments, each of which is associated with a particular functional unit of the multiple functional units. At any point in time, the register file segments as allocated to each functional unit each contain the same content. A multi-ported register file is typically metal limited to the area consumed by the circuit proportional with the square of the number of ports. The processor **100** has a register file structure divided into a plurality of separate and independent register files to form a layout structure with an improved layout efficiency. The read ports of the total register file structure **216** are allocated among the separate and individual register files. Each of the separate and individual register files has write ports that correspond to the total number of write ports in the total register file structure. Writes are fully broadcast so that all of the separate and individual register files are coherent.

The media functional units **220** are multiple single-instruction-multiple-datapath (MSIMD) media functional units. Each of the media functional units **220** is capable of processing parallel 16-bit components. Various parallel 16-bit operations supply the single-instruction-multiple-datapath capability for the processor **100** including add, multiply-add, shift, compare, and the like. The media functional units **220** operate in combination as tightly coupled digital signal processors (DSPs). Each media functional unit **220** has an separate and individual sub-instruction stream, but all three media functional units **220** execute synchronously so that the subinstructions progress lock-step through pipeline stages.

The general functional unit **222** is a RISC processor capable of executing arithmetic logic unit (ALU) operations, loads and stores, branches, and various specialized and esoteric functions such as parallel power operations, reciprocal square root operations, and many others. The general functional unit **222** supports less common parallel operations such as the parallel reciprocal square root instruction.

The illustrative instruction cache **210** is two-way set-associative, has a 16 Kbyte capacity, and includes hardware support to maintain coherence, allowing dynamic optimizations through self-modifying code. Software is used to indicate that the instruction storage is being modified when modifications occur. The 16K capacity is suitable for performing graphic loops, other multimedia tasks or processes, and general-purpose Java™ code. Coherency is maintained by hardware that supports write-through, non-allocating caching. Self-modifying code is supported through explicit use of "store-to-instruction-space" instruction *store2i*. Software uses the *store2i* instruction to maintain coherency with the instruction cache **210** so that the instruction caches **210** do not have to be snooped on every single store operation issued by the media processing unit **110**.

The pipeline control unit **226** is connected between the instruction buffer **214** and the functional units

and schedules the transfer of instructions to the functional units. The pipeline control unit **226** also receives status signals from the functional units and the load/store unit **218** and uses the status signals to perform several control functions. The pipeline control unit **226** maintains a scoreboard, generates stalls and bypass controls. The pipeline control unit **226** also generates traps and maintains special registers.

Each media processing unit **110** and **112** includes a split register file **216**, a single logical register file including 128 thirty-two bit registers. The split register file **216** is split into a plurality of register file segments **224** to form a multi-ported structure that is replicated to reduce the integrated circuit die area and to reduce access time. A separate register file segment **224** is allocated to each of the media functional units **220** and the general functional unit **222**. In the illustrative embodiment, each register file segment **224** has 128 32-bit registers. The first 96 registers (0-95) in the register file segment **224** are global registers. All functional units can write to the 96 global registers. The global registers are coherent across all functional units (MFU and GFU) so that any write operation to a global register by any functional unit is broadcast to all register file segments **224**. Registers 96-127 in the register file segments **224** are local registers. Local registers allocated to a functional unit are not accessible or "visible" to other functional units.

The media processing units **110** and **112** are highly structured computation blocks that execute software-scheduled data computation operations with fixed, deterministic and relatively short instruction latencies, operational characteristics yielding simplification in both function and cycle time. The operational characteristics support multiple instruction issue through a pragmatic very large instruction word (VLIW) approach that avoids hardware interlocks to account for software that does not schedule operations properly. Such hardware interlocks are typically complex, error-prone, and create multiple critical paths. A VLIW instruction word always includes one instruction that executes in the general functional unit (GFU) **222** and from zero to three instructions that execute in the media functional units (MFU) **220**. A MFU instruction field within the VLIW instruction word includes an operation code (opcode) field, three source register (or immediate) fields, and one destination register field.

Instructions are executed in-order in the processor **100** but loads can finish out-of-order with respect to other instructions and with respect to other loads, allowing loads to be moved up in the instruction stream so that data can be streamed from main memory. The execution model eliminates the usage and overhead resources of an instruction window, reservation stations, a re-order buffer, or other blocks for handling instruction ordering. Elimination of the instruction ordering structures and overhead resources is highly advantageous since the eliminated blocks typically consume a large portion of an integrated circuit die. For example, the eliminated blocks consume about 30% of the die area of a Pentium II processor.

To avoid software scheduling errors, the media processing units **110** and **112** are high-performance but simplified with respect to both compilation and execution. The media processing units **110** and **112** are most generally classified as a simple 2-scalar execution engine with full bypassing and hardware interlocks on load

operations. The instructions include loads, stores, arithmetic and logic (ALU) instructions, and branch instructions so that scheduling for the processor **100** is essentially equivalent to scheduling for a simple 2-scalar execution engine for each of the two media processing units **110** and **112**.

The processor **100** supports full bypasses between the first two execution units within the media processing unit **110** and **112** and has a scoreboard in the general functional unit **222** for load operations so that the compiler does not need to handle nondeterministic latencies due to cache misses. The processor **100** scoreboards long latency operations that are executed in the general functional unit **222**, for example a reciprocal square-root operation, to simplify scheduling across execution units. The scoreboard (not shown) operates by tracking a record of an instruction packet or group from the time the instruction enters a functional unit until the instruction is finished and the result becomes available. A VLIW instruction packet contains one GFU instruction and from zero to three MFU instructions. The source and destination registers of all instructions in an incoming VLIW instruction packet are checked against the scoreboard. Any true dependencies or output dependencies stall the entire packet until the result is ready. Use of a scoreboarded result as an operand causes instruction issue to stall for a sufficient number of cycles to allow the result to become available. If the referencing instruction that provokes the stall executes on the general functional unit **222** or the first media functional unit **220**, then the stall only endures until the result is available for intra-unit bypass. For the case of a *load* instruction that hits in the data cache **106**, the stall may last only one cycle. If the referencing instruction is on the second or third media functional units **220**, then the stall endures until the result reaches the writeback stage in the pipeline where the result is bypassed in transmission to the split register file **216**.

The scoreboard automatically manages load delays that occur during a load hit. In an illustrative embodiment, all loads enter the scoreboard to simplify software scheduling and eliminate NOPs in the instruction stream.

The scoreboard is used to manage most interlock conditions between the general functional unit **222** and the media functional units **220**. All loads and non-pipelined long-latency operations of the general functional unit **222** are scoreboarded. The long-latency operations include division *idiv*, *fdiv* instructions, reciprocal square root *frecsqrt*, *precsqrt* instructions, and power *ppower* instructions. None of the results of the media functional units **220** is scoreboarded. Non-scoreboarded results are available to subsequent operations on the functional unit that produces the results following the latency of the instruction.

The illustrative VLIW processor **100** executes a plurality of instructions in parallel on the plurality of independent processors using low thread synchronization overhead to operate with the same level of performance as an increased width VLIW processor. For applications with a high level of instruction-level parallelism, the VLIW processor **100** utilizes the two independent four-wide VLIW processors, media processing units **110** and **112**, to selectively operate either as an eight-wide VLIW processor or two independent four-wide VLIW processors that mutually execute separate threads. The independent media processing units **110** and **112** each

includes a very rich set of functional units to form, in combination, a highly powerful processor without the complexity of implementing the extensive control circuitry and connections of an eight-wide VLIW processor.

The VLIW processor **100** has a rendering rate of over fifty million triangles per second without accounting for operating system overhead. To exploit the parallelism of the VLIW processor **100** when processing independent data constructs, such as triangles, the media processing units **110** and **112** simply operate on multiple elements (vertices) in parallel so that the utilization rate of the functional units is high. The result is effectively a spatial software pipelining. Instead of overlapping several loop iterations in time, as is practiced in conventional VLIW processors, the illustrative VLIW processor **100** overlaps operations by executing computations on a plurality of vertices simultaneously. For other types of operations that have high instruction-level parallelism, high trip count loops are software-pipelined to fully utilize the media functional units **220**.

The usage of media processing units **110** and **112** in parallel to selectively operate either in coordination as an increased-width VLIW processor or independently as a multiple-thread processor is highly advantageous for efficiently performing applications that have differing levels of parallelism. Parallelism is rarely found beyond the level of about four instructions, or even two instructions, per cycle in general purpose code. Therefore, the illustrative processor **100** includes two media processing units **110** and **112** that are selectively operated independently. Therefore, the two four-wide VLIW media processing units **110** and **112** typically operates on two independent threads, rather than wasting most of the eight functional units in an eight-wide processor, possibly attaining twice the performance of a much wider VLIW processor executing at the same clock rate. The usage of independent media processing units **110** and **112** is particularly advantageous for Java™ applications which often have multiple threads of execution.

Therefore, data feeding specifications of the processor **100** are far beyond the capabilities of cost-effective memory systems. Sufficient data bandwidth is achieved by rendering of compressed geometry using the geometry decompressor **104**, an on-chip real-time geometry decompression engine. Data geometry is stored in main memory in a compressed format. At render time, the data geometry is fetched and decompressed in real-time on the integrated circuit of the processor **100**. The geometry decompressor **104** advantageously saves memory space and memory transfer bandwidth. The compressed geometry uses an optimized generalized mesh structure that explicitly calls out most shared vertices between triangles, allowing the processor **100** to transform and light most vertices only once. In a typical compressed mesh, the triangle throughput of the transform-and-light stage is increased by a factor of four or more over the throughput for isolated triangles.

Referring to **FIGURE 4**, a schematic block diagram illustrates an embodiment of the split register file **216** that is suitable for usage in the processor **100**. The split register file **216** supplies all operands of processor instructions that execute in the media functional units **220** and the general functional units **222** and receives results of the instruction execution from the execution units. The split register file **216** operates as an interface to the geometry decompressor **104** and supplies data to the setup/draw unit **108**. The split register file **216** is the

source and destination of store and load operations, respectively.

In the illustrative processor **100**, the split register file **216** in each of the media processing units **110** has 256 registers. Graphics processing places a heavy burden on register usage. Therefore, a large number of registers is supplied by the split register file **216** so that performance is not limited by loads and stores or handling of intermediate results including graphics "fills" and "spills". The illustrative split register file **216** includes twelve read ports and five write ports, supplying total data read and write capacity between the central registers of the split register file **216** and all media functional units **220** and the general functional unit **222**. Total read and write capacity promotes flexibility and facility in programming both of hand-coded routines and compiler-generated code.

Large, multiple-ported register files are typically metal-limited so that the register area is proportional with the square of the number of ports. A sixteen port file is roughly proportional in size and speed to a value of 256. The illustrative split register file **216** is divided into four register file segments **310**, **312**, **314**, and **316**, each having three read ports and four write ports so that each register file segment has a size and speed proportional to 49 for a total area for the four segments that is proportional to 196. The total area is therefore potentially smaller and faster than a single central register file. Write operations are fully broadcast so that all files are maintained coherent. Logically, the split register file **216** is no different from a single central register file However, from the perspective of layout efficiency, the split register file **216** is highly advantageous, allowing for reduced size and improved performance.

The new media data that is operated upon by the processor **100** is typically heavily compressed. Data transfers are communicated in a compressed format from main memory and input/output devices to pins of the processor **100**, subsequently decompressed on the integrated circuit holding the processor **100**, and passed to the split register file **216**.

Splitting the register file into multiple segments in the split register file **216** in combination with the character of data accesses in which multiple bytes are transferred to the plurality of execution units concurrently, results in a high utilization rate of the data supplied to the integrated circuit chip and effectively leads to a much higher data bandwidth than is supported on general-purpose processors. The highest data bandwidth requirement is therefore not between the input/output pins and the central processing units, but is rather between the decompressed data source and the remainder of the processor. For graphics processing, the highest data bandwidth requirement is between the geometry decompressor **104** and the split register file **216**. For video decompression, the highest data bandwidth requirement is internal to the split register file **216**. Data transfers between the geometry decompressor **104** and the split register file **216** and data transfers between various registers of the split register file **216** can be wide and run at processor speed, advantageously delivering a large bandwidth.

The register file **216** is a focal point for attaining the very large bandwidth of the processor **100**. The processor **100** transfers data using a plurality of data transfer techniques. In one example of a data transfer technique, cacheable data is loaded into the split register file **216** through normal load operations at a low rate of up to eight bytes per cycle. In another example, streaming data is transferred to the split register file **216** through group load operations which transfer thirty-two bytes from memory directly into eight consecutive 32-bit registers. The processor **100** utilizes the streaming data operation to receive compressed video data for decompression.

Compressed graphics data is received via a direct memory access (DMA) unit in the geometry decompressor **104**. The compressed graphics data is decompressed by the geometry decompressor **104** and loaded at a high bandwidth rate into the split register file **216** via group load operations that are mapped to the geometry decompressor **104**.

Data such as transformed and lit vertices are transferred to the setup/draw unit **108** through store pair instructions which can accommodate eight bytes per cycle mapped to the setup/draw unit **108**.

Load operations are non-blocking and scoreboarded so that a long latency inherent to loads can be hidden by early scheduling.

General purpose applications often fail to exploit the large register file **216**. Statistical analysis shows that compilers do not effectively use the large number of registers in the split register file **216**. However, aggressive in-lining techniques that have traditionally been restricted due to the limited number of registers in conventional systems may be advantageously used in the processor **100** to exploit the large number of registers in the split register file **216**. In a software system that exploits the large number of registers in the processor **100**, the complete set of registers is saved upon the event of a thread (context) switch. When only a few registers of the entire set of registers is used, saving all registers in the full thread switch is wasteful. Waste is avoided in the processor **100** by supporting individual marking of registers. Octants of the thirty-two registers can be marked as "dirty" if used, and are consequently saved conditionally.

In various embodiments, the split register file **216** is leveraged by dedicating fields for globals, trap registers, and the like.

Referring to **FIGURE 5A**, a schematic block diagram shows a logical view of the register file **216** and functional units in the processor **100**. The physical implementation of the core processor **100** is simplified by replicating a single functional unit to form the three media processing units **110**. The media processing units **110** include circuits that execute various arithmetic and logical operations including general-purpose code, graphics code, and video-image-speech (VIS) processing. VIS processing includes video processing, image processing, digital signal processing (DSP) loops, speech processing, and voice recognition algorithms, for example.

Referring to **FIGURE 5B**, a schematic block diagram illustrates a logical view of a combination of two independent processors to form a wide-VLIW processor **100**. Each of the illustrative VLIW processors includes an instruction buffer **214**, a register file **216**, and four functional units arranged in a group of three media functional units **220**, and one general functional unit **222**. The instruction buffer **214** in each of the independent processors supplies up to four subinstructions to the register file **216**. The up to four subinstructions execute on the media functional units **220** and the general functional unit **222**. For applications with a high level of instruction-level parallelism, the eight-wide VLIW processor **100** executes a plurality of instructions in parallel on the plurality of independent processors using low thread synchronization overhead to operate with the same level of performance as an increased width VLIW processor. The VLIW processor **100** includes two independent four-wide VLIW processors **110** and **112** to selectively operate either as an eight-wide VLIW processor or two independent four-wide VLIW processors that mutually execute separate threads. The independent processors includes a very rich set of functional units to form, in combination, a highly powerful processor, without the complexity of implementing the extensive control circuitry and connections of the eight-wide VLIW processor.

Referring to **FIGURE 6**, a simplified pictorial schematic diagram depicts an example of instruction execution among a plurality of media functional units **220**. Results generated by various internal function blocks within a first individual media functional unit are immediately accessible internally to the first media functional unit **510** but are only accessible globally by other media functional units **512** and **514** and by the general functional unit five cycles after the instruction enters the first media functional unit **510**, regardless of the actual latency of the instruction. Therefore, instructions executing within a functional unit can be scheduled by software to execute immediately, taking into consideration the actual latency of the instruction. In contrast, software that schedules instructions executing in different functional units is expected to account for the five cycle latency. In the diagram, the shaded areas represent the stage at which the pipeline completes execution of an instruction and generates final result values. A result is not available internal to a functional unit a final shaded stage completes. In the example, media processing unit **110** instructions have three different latencies - four cycles for instructions such as fmuladd and fadd, two cycles for instructions such as pmuladd, and one cycle for instructions like padd and xor.

Although internal bypass logic within a media functional unit **220** forwards results to execution units within the same media functional unit **220**, the internal bypass logic does not detect incorrect attempts to reference a result before the result is available.

Software that schedules instructions for which a dependency occurs between a particular media functional unit, for example **512**, and other media functional units **510** and **514**, or between the particular media functional unit **512** and the general functional unit **222**, is to account for the five cycle latency between entry of an instruction to the media functional unit **512** and the five cycle pipeline duration.

Referring to **FIGURE 7**, a simplified schematic timing diagram illustrates timing of the processor pipeline **600**. The pipeline **600** includes nine stages including three initiating stages, a plurality of execution phases, and two terminating stages. The three initiating stages are optimized to include only those operations necessary for decoding instructions so that jump and call instructions, which are pervasive in the Java™ language, execute quickly. Optimization of the initiating stages advantageously facilitates branch prediction since branches, jumps, and calls execute quickly and do not introduce many bubbles.

The first of the initiating stages is a fetch stage **610** during which the processor **100** fetches instructions from the 16Kbyte two-way set-associative instruction cache **210**. The fetched instructions are aligned in the instruction aligner **212** and forwarded to the instruction buffer **214** in an align stage **612**, a second stage of the initiating stages. The aligning operation properly positions the instructions for storage in a particular segment of the four register file segments **310, 312, 314**, and **316** and for execution in an associated functional unit of the three media functional units **220** and one general functional unit **222**. In a third stage, a decoding stage **614** of the initiating stages, the fetched and aligned VLIW instruction packet is decoded and the scoreboard (not shown) is read and updated in parallel. The four register file segments **310, 312, 314**, and **316** each holds either floating-point data or integer data. The register files are read in the decoding (D) stage.

Following the decoding stage **614**, the execution stages are performed. The two terminating stages include a trap-handling stage **660** and a write-back stage **662** during which result data is written-back to the split register file **216**.

Referring to **FIGUREs 8A, 8B**, and **8C** respectively, a schematic block diagram shows an embodiment of the general functional unit **222**, a simplified schematic timing diagram illustrating timing of general functional unit pipelines **700**, and a bypass diagram showing possible bypasses for the general functional unit **222**. The general functional unit **222** supports instructions that execute in several different pipelines. Instructions include single-cycle ALU operations, four-cycle getir instructions, and five-cycle setir instructions. Long-latency instructions are not fully pipelined. The general functional unit **222** supports six-cycle and 34-cycle long operations and includes a dedicated pipeline for load/store operations.

The general functional unit **222** and pipeline control unit **226**, in combination, include four pipelines, Gpipe1 **750**, Gpipe2 **752**, Gpipe3 **754**, and a load/store pipeline **756**. The load/store pipeline **756** and the Gpipe1 **750** are included in the pipeline control unit **226**. The Gpipe2 **752** and Gpipe3 **754** are located in the general functional unit **222**. The general functional unit **222** includes a controller **760** that supplies control signals for the pipelines Gpipe1 **750**, Gpipe2 **752**, and Gpipe3 **754**. The pipelines include execution stages (En) and annex stages (An).

Results from instructions executed by the general functional unit **222** and media functional units **220** are not immediately written to the register file segments **224**. The results are staged in the annexes and broadcast to

the register file segments **224** in T-stage if no trap is present. The register file segments **224** latch the results locally and update the registers in the next clock cycle. The annex contains destination rd specifiers of all instructions from the A1-stage and onward. The annex maintains valid bits for each stage of the pipeline. The annex also contains priority logic to determine the most recent value of a register in the register file.

The general functional unit pipelines **700** include a load pipeline **710**, a 1-cycle pipeline **712**, a 6-cycle pipeline **712**, and a 34-cycle pipeline **714**. Pipeline stages include execution stages (E and En), annex stages (An), trap-handling stages (T), and write-back stages (WB). Stages An and En are prioritized with smaller priority numbers n having a higher priority.

The processor **100** supports precise traps. Precise exceptions are detected by E4/A3 stages of media functional unit and general functional unit operations. One-cycle operations are stages in annex and trap stages (A1, A2, A3, T) until all exceptions in one VLIW group are detected. Traps are generated in the trap-generating stages (T). When the general functional unit **222** detects a trap in a VLIW group, all instructions in the VLIW group are canceled.

When a long-latency operation is in the final execute stage (E6 stage for the 6-cycle pipeline **712** or E34 stage for the 34-cycle pipeline **714**), and a valid instruction is under execution in the A3-stage of the annex, then the long-latency instruction is held in a register, called an A4-stage register, inside the annex and is broadcast to the register file segments **224** only when the VLIW group under execution does not include a one-cycle GFU instruction that is to be broadcast.

Results of long-latency instructions are bypassed to more recently issued GFU and MFU instructions as soon as the results are available. For example, results of a long-latency instruction are bypassed from the E6-stage of a 6-cycle instruction to any GFU and MFU instruction in the decoding (D) stage. If a long-latency instruction is stalled by another instruction in the VLIW group, results of the stalled long-latency instruction are bypassed from the annex (A4) stage to all instructions in the general functional unit **222** and all media functional units **220** in the decoding (D) stage.

Data from the T-stage of the pipelines are broadcast to all the register file segments **224**, which latch the data in the writeback (WB) stage before writing the data to the storage cells.

The bypass diagram shown in **FIGURE 8C** shows operation of the bypasses in the general functional unit **222**. The pipelines **700** are arranged in levels, shown arranged from right to left, with level_1 having a higher priority than level_2 and so on. Among the stages in the same level, no difference in priority is enforced. For the execution stages (E), if a load is followed by a gfu instruction with the same destination specifier rd, then the higher priority gfu instruction enters the E-stage when the load is in a ldx1 stage of an annex (A). The annexes include compare-match logic that enforce the bypass priority among instructions.

The diagram assumes that the load returns in cycle 3 and enters ldx1-stage in cycle 4 and ldx2-stage in cycle 5. Inst1 is unstalled in cycle 4. In cycle 5, when inst2 is in D-stage, the only bypass younger than the load in ldx2-stage is in the bypass from E-stage of gfu/mfu1 instructions. Therefore, ldx2-stage is one level below (level_2) the E-stage of the gfu/mfu1 instructions.

Similarly, in cycle 6, when the load is in ldx3-stage, the stages younger than ldx_3 stage but having the same destination specifiers rd are E/A1 stages of gfu and mfu1 instructions. Therefore, ldx_3 stage is one level below (level_3) the A1-stages of gfu/mfu1 instructions in bypass. Similarly, ldx4_stage is one level below A2-stage (level_4) in bypass priority. When the load is in ldx1-stage (in cycle 4), no younger instructions with the same rd specifier as the load are possible due to hardware interlock with the load. Therefore, ldx1-stage has the highest priority (level_1) in bypass.

Priority is similarly enforced for long latency instructions.

The annexes include multiplexers that select matching stages among the bypass levels in a priority logic that selects data based on priority matching within a priority level. The annexes include compare logic that compares destination specifiers rd against each of the specifiers rs1, rs2, and rd.

In the illustrative embodiment, the bypass logic for the first media functional unit **220** mfu1 is the same as the bypass logic for the general functional unit **222** since full bypass between the gfu and mfu1 is supported. Annexes for the second and third media functional units **220** (mfu2 and mfu3) are mutually identical but simpler than the gfu/mfu1 bypass logic since full bypass is not supported for mfu2/mfu3. In other embodiments, full bypass may be implemented in all positions.

Referring to **FIGURE 9**, a simplified schematic timing diagram illustrates timing of media functional unit pipelines **800** including single precision and integer pipelines. The media functional units **220** execute single-precision, double-precision, and long operations. The media functional units **220** are connected to the register file segments **224** via 32-bit read ports so that two cycles are used to read the source operands for 64-bit double-precision and long operations. A "pair instruction" is any MFU instruction that utilizes a second instruction to either read or write the lower or higher order pair of a source operand or destination operand. The second instruction is not part of the instruction stream from the instruction memory but rather is generated in hardware. The media functional units **220** support 1-cycle **810**, 2-cycle **812**, and 4-cycle **814** instructions on single precision and integer operations.

The media functional units **220** execute pair instructions including two-cycle latency instructions that read two long integers or two doubles and generate a single integer result, two-cycle latency instructions that read two long integers and write to a long integer, five-cycle latency instructions that read two double or one double operands and generate a double precision result, and seven-cycle latency instructions that read two double

precision operands and generate a double-precision result. When a five cycle or seven cycle latency instruction is included in a VLIW group, all other instructions in the group are stalled in E-stage so that all instructions in the VLIW group read T-stage simultaneously: Precise exceptions are thus supported even for five-cycle and seven-cycle latency instructions.

When a second instruction of a pair is generated by hardware for a media functional unit **220**, instructions in a next issued VLIW group are stalled for one cycle so that the second instruction of the pair does not have a conflict with the instruction in the next VLIW group for writing to the register file segment **224**.

For efficient operation, when a VLIW group contains one or more pair instructions and the next subsequent VLIW group in a sequence is vacant in the pair instruction positions, then the instructions in the subsequent VLIW group in the nonvacant positions are executed along with the initial VLIW group. A group position is vacant when no instruction is included or the instruction is a 'nop', any instruction which writes to register 0 and has no side effects. The pipeline control unit **226** detects a vacant position in a VLIW group by analyzing the opcode and the destination register rd of the instruction.

Referring to **FIGUREs 10A-10C**, an instruction sequence table and two pipeline diagrams illustrate execution of VLIW groups including a five-cycle pair instruction and a seven-cycle pair instruction. **FIGURE 10A** is an instruction sequence table showing five VLIW groups (VLIW_n) and instructions executed in the media functional units **220** (MFU3, MFU2, and MFU1) and in the general functional unit **222** (GFU) for sequentially issued instruction groups n=1 to 5.

**FIGURE 10B** shows instruction mfu1_1 in the MFU1 position of the group vliw_1 as a five-cycle latency pair instruction and instruction mfu1_2 in the MFU1 position of the group vliw_2 as a valid instruction and not a 'nop' so that instruction gfu_1 in the GFU position of the group vliw_1 is stalled in the E-stage for one cycle. Stalling of the instruction gfu_1 causes gfu_1 to reach the T-stage at the same time as the instruction mfu1_1. The first and second inherent instruction of a pair instruction are atomic instructions between which no traps or interrupts can occur. All errors or exceptions caused by the double or long pair instructions are detected when the instruction reaches the final Execute (E4) stage. If an error or exception is detected in the E4-stage of the pair instruction, both the first and inherent second instruction of the pair are canceled.

In the illustrative example shown in **FIGURE 10B**, if a trap occurs in cycle 7, both the first and inherent second instruction of mfu1_1 are canceled. If a trap occurs in cycle 8 due to external interrupt, asynchronous error, instructions in group vliw_2, or the like, then the inherent second instruction of mfu1_1 is allowed to finish execution and only the instructions in group vliw_2 are canceled.

**FIGURE 10C** shows instruction mfu1_1 in the MFU1 position of the group vliw_1 as a seven-cycle latency pair instruction and no pair instructions in group vliw_2. The gfu_1 instruction in the group containing

the seven-cycle latency pair stalls for four cycles in the E-stage.

Referring to **FIGUREs 11A-11C**, pipeline diagrams show examples illustrating synchronization of pair instructions in a group. If a two-cycle latency pair instruction is included in group vliw_1, then group vliw_2 stalls in D-stage for a cycle if the position in group vliw_2 corresponding to the two-cycle latency pair instruction is not vacant.

If a five-cycle or seven-cycle latency pair instruction is included in group vliw_1, other instructions in group vliw_1 stall in E-stage so that all instructions in group vliw_1 read the trap-generating (T) stage simultaneously.

Instructions in group vliw_2 stall for an additional cycle in E-stage only if a port conflict of the inherent second instruction of a pair and an instruction in group vliw_2 is scheduled to occur.

In the example shown in **FIGURE 11A**, instruction mfu1_1 is a seven-cycle latency pair instruction and instruction mfu2_2 in group vliw_2 is a five-cycle latency pair instruction.

In the example shown in **FIGURE 11B**, in group vliw_1 instruction mfu1_1 is a two-cycle latency pair instruction, instruction mfu2_1 is a five-cycle latency pair instruction, and instruction mfu3_1 is a seven-cycle latency pair instruction. Group vliw_2 includes at least one valid MFU instruction which is not a five-cycle or seven-cycle latency pair instruction.

In the example shown in **FIGURE 11C**, in group vliw_1 instruction mfu2_1 is a five-cycle latency pair instruction and instruction mfu3_1 is a seven-cycle latency pair instruction, but instruction mfu1_1 is not a two-cycle latency pair instruction. The difference between the examples shown in **FIGURE 11B** and **FIGURE 11C** is that instructions in group vliw_2 in the latter case stall in the E-stage in cycle 6 to avoid port conflicts of the inherent second instructions of instructions mfu2_1 and mfu3_1 with mfu2_2 and mfu3_2, respectively.

Referring to **FIGUREs 12A**, **12B**, **12C**, and **12D**, respective schematic block diagrams illustrate the pipeline control unit **226** segments allocated to all of the functional units GFU, MFU1, MFU2, and MFU3. The pipeline control unit **226** imposes several scheduling rules that apply to bypass between instructions in a single VLIW group. Full bypass is implemented between instructions executed by functional units GFU and MFU1 so that bypass rules are identical for bypass from results of pair instructions in MFU1 to more recently issued instructions executed in the GFU and MFU1 functional units. For other cases, an additional one cycle penalty is imposed for bypass from a pair instruction to more recently issued instructions in other groups. The scheduling rules are imposed by control units allocated to the general functional unit **222** and the media functional units **220**. A pcu_gf control unit (pcu_gf_ctl **1110**) is the control block for instructions executing in the general functional unit **222**. Similarly, pcu_mf1_ctl **1120**, pcu_mf2_ctl **1122**, and pcu_mf3_ctl **1124** are control blocks for mfu1, mfu2, and mfu3, respectively. The pcu/ functional unit control blocks generate D-stage stalls, generate D-stage

bypasses for "alu_usc_immediate" cases and for generating multiplexer select signals for E-stage bypasses. The control blocks for the various functional units are positioned adjacent to the scoreboard datapath associated to the particular functional unit. The pcu control units include a partial decoder, such as gfu partial decoder **1130** and mfu partial decoders **1132**, **1134**, and **1136**.

The pipeline control unit **226** also include a plurality of internal registers (ir), many of which are not accessible by a user. One internal register of the pipeline control unit **226** is a processor control register (PCR) that controls power management, instruction and data cache enables, pipeline enable, and branch predict taken enable.

The pcu control units perform various functions including qualifying scoreboard hits with immediate bits, sending operation type signals to the load/store unit **218**, and handling various instructions including getir, setir, sethi, jmpl, membar, and prefetch. Signals generated by the decoder include a gfu_imm signal that designates whether source rs2 is immediate, a gfu_load signal that designates whether a gfu instruction is a load, a gfu_ldg signal that identifies whether the instruction is a group load, and a gfu_ldpair signal that designates whether the instruction is a paired instruction within a load pair. Generated signals further include a gfu_store signal that identifies a store instruction, a gf_stpair signal the indicates whether the gfu instruction is a store pair instruction, and a gfu_cas signal which indicates that the gfu instruction is a cas instruction. A gfu_prefetch signal indicates the gfu instruction is a prefetch. A gfu_call signal designates a call instruction with r2 as a destination specifier. A gfu_branch signal designates a branch instruction with the rd field as a source specifier. The gfu_nop signal designates a nop. A gfu_illegal signal identifies an illegal instruction. A gfu_privilege signal designates a privileged instruction. A gfu_sir signal indicates a software initiated reset instruction. A gfu_softrap signal identifies a softrap instruction. Signals including gfu_sethi, gfu_setlo, and gfu_addlo designate sethi instructions. A gfu_long signal indicates a long latency instruction. Signals including gfu_setir, gfu_getir, gfu_setir_psr, and gfu_memissue respectively designate setir, getir, setir to PSR, and membar instructions.

The pcu_gf_ctl **1110** generates D-stage and E-stage stalls of the general functional unit **222**, generates signals to hold the D-stage of the gfu instruction, source, and destination operands.

The pcu_gf_ctl **1110** controls full bypass between the general functional unit **222** (gfu) and the media functional unit **220** (mfu1). The pcu_gf_ctl **1110** generates bypass signals in several circumstances. An ALU use immediate bypass is generated if any of the source specifiers of the gfu instruction depends on the results of an immediately preceding 1 cycle gfu or mfu1 instruction. If a source specifier of any gfu instruction in E-stage awaits load data, then the pcu_gf_ctl **1110** asserts appropriate select signals to select the data returning from either the load/store unit **218** or the data cache **106**. If the source specifier of any gfu instruction in D-stage is dependent on a previous long latency instruction, then the pcu_gf_ctl **1110** asserts appropriate select signals to select the long latency data. If an E-stage stall occurs and any source operand is not dependent on a load data

return, then the pcu_gf_ctl **1110** asserts appropriate signals to hold the data the source operand has already bypassed.

The pcu_mf1_ctl **1120** is similar to the pcu_gf_ctl **1110** and performs functions including partial decoding of the mfu1 instruction to supply and maintain the D-stage opcode of the mfu1 instructions. The pcu_mf1_ctl **1120** generates all stalls of the mfu1 instruction and recirculating the D-stage mfu1 instruction. The pcu_mf1_ctl **1120** generates bypass selects for mfu1 instructions and sends load dependency information to the mfu1 annex so that the annex selects a proper bypass if the instruction is stalled in D-stage with load dependency. The pcu_mf1_ctl **1120** detects bypasses for ALU-use immediate cases and generates the inherent second instruction of a paired mfu1 instruction. The mf1 also generates synchronizing stalls for mfu instructions.

In the illustrative embodiment, pcu_mf2_ctl **1122** and pcu_mf3_ctl **1124**, control blocks for mfu2 and mfu3 instructions, are the identical but differ from pcu_mf1_ctl **1120** because full bypass is not supported between mfu2/mfu3 and gfu.

A first rule applies for a VLIW group N that contains a five-cycle or seven-cycle latency pair instruction of the format 'pair ax, bx, cx' and the inherent second instruction of the pair has the format 'helper ay, by, cy' where ax/ay, bx/by, and cx/cy are even-odd pairs. If (1) at least one pair instruction is included in either of the next two VLIW groups N+1 or N+2, or (2) a valid MFUx instruction in the VLIW group N+1 is included in the position corresponding to the position of a pair instruction in the VLIW group N. then (i) any more recently issued pair instruction within the same functional unit is to be at least four groups apart (VLIW group N+4) to bypass the results of the pair instruction in VLIW group N, and (ii) any more recently issued instruction that uses the results cx/cy is to be at least four groups apart (VLIW group N+4). Otherwise, any more recently issued instruction that uses the results of the pair instruction is to be at least five groups apart (VLIW group N+5).

A second rule applies when a VLIW position holds a pair instruction in VLIW group N and a vacancy in VLIW group N+1. Instructions in VLIW group N+1 are not to write to the same destination register rd as the inherent second instruction of the pair in VLIW group N.

A third rule applies for a two-cycle latency pair instruction that generates integer results (dcmp and lcmp instructions) in a VLIW group N. Instructions in VLIW group N+1 can bypass results of the dcmp and lcmp operations.

A fourth rule applies for a two-cycle latency pair instruction that generates a 64-bit result (ladd/lsub) in a VLIW group N. Any instruction in VLIW group N+1 can bypass the results of the pair instruction in VLIW group N.

A fifth rule specifies that an assembler is expected to schedule the correct even and odd register specifiers since pipeline control unit **226** logic does not check to determine whether the even or odd register

specifiers are correct. Pipeline control unit **226** logic reads the source operands that are specified in the instruction and sends the source operands to the functional unit for execution. For pair instructions, logic in the pipeline control unit **226** then inverts bit[0] of the source operands and sends the newly determined source operands to the functional unit for execution in the next cycle. The results returned by the functional unit after N-cycle, where N is the latency of the pair instruction, are written into the rd specifier specified in the instruction. Results returned in cycle N+1 are written to the rd specifier with bit[0] inverted. If a pair instruction has any of the source or destination operands specified as r0, then the source/destination operands of the inherent second instruction of a pair are also r0.

The pipeline control unit **226** supports full bypass between the general functional unit **222** and MFU1 of the media functional units **220**. Thus results of instructions executed in MFU1 are available in the same cycle to instructions in the D-stage in GFU and MFU1 units. However, results of instructions executed in MFU2 and MFU3 are available to the GFU and MFU1 functional units only after results enter the T-stage. Specifically, a GFU instruction that uses the result of a one, two or four-cycle MFU2 instruction has to be at least five cycles later. GFU and MFU1 instructions have a two-cycle best case load-use penalty. MFU2 and MFU3 instructions have a three-cycle best case load-use penalty. A GFU instruction having an output dependency with a previous load and the load is a data cache hit returning data in the A1-stage has a three cycle penalty.

All pipeline stages from which the source operands of a GFU instruction bypass data are maintained in a pipeline control unit - general functional unit interface **1110** shown in **FIGURE 12A**. Similar interfaces are included for each of the media functional units **220**, MFU1, MFU2, and MFU3 shown in **FIGUREs 12B, 12C,** and **12C**, respectively.

Referring to **FIGURE 13**, a schematic block diagram illustrates a load annex block **1200** within the pipeline control unit **226**. The load annex block **1200** includes a storage **1210** for holding data, read specifiers, data size, and a valid bit. Read data is stored in a separate FIFO **1212** in the annexes. The read data is simply shifted down every cycle. Every cycle, stage bits are shifted to the right by one position.

If a trap or flush occurs in a cycle N, all entries with a nonzero count are invalidated before the entries are shifted down in cycle N+1.

Load addresses are calculated in E-stage of the pipeline. The data cache **106** is accessed in C/A1-stage of the pipeline. If the load hits the data cache **106**, data returns from the data cache **106** in the same cycle. If the load misses the data cache **106**, data returns from the load/store unit **218** after the load/store unit receives data from either the main memory (not shown) or interface (not shown). Data from the load/store unit **218** is received in any of annex (A1)- write-back (WB) stages or the write-back stage (WB).

Load data enters the annexes and is staged in the annexes for three cycles before being broadcast to the

register file **216**. The load operation is staged in the ldx1-ldx4 stages in the annex. By staging the load for three cycles, all precise exceptions caused by either the load or other instructions in the same VLIW group cause cancellation of the load. During the ldx4 stage, the load is written to the register file **216**. Data cannot be accessed and written into the register file **216** in the same cycle so an additional stage is inserted to hold the data while the data is written.

When load data enters the annex, the age of the data is indicated by stage bits (A1-T). If a trap is detected before the load reaches the write-back (WB) stage, the load is invalidated.

Once the data returns from the load/store unit **218**, the data enters a ldx FIFO (not shown) in all the annexes. The annex has four entries. Since the 64-bit write port in each of the register file segment **224** is dedicated for loads, the FIFO is shifted down one entry each clock cycle.

The scoreboard is a storage structure that maintains information regarding unfinished loads and long latency operations in the general functional unit **222**. The scoreboard allows in-order processing with non-blocking memory. The scoreboard supplies a hardware interlock between any unfinished load or long-latency operation and a more recently issued instruction that has data/output dependency with the load or long-latency operation.

The scoreboard has separate entries for loads and long-latency operations. When a new instruction enters the decode stage, the scoreboard compares the source operand rs and destination operand rd specifiers with all entries.

The scoreboard has an E-stage entry for holding information relating to loads and long-latency operations when the operations transition from the D-stage to the E-stage. The E-stage entry includes a field for storing the destination register specifier of the load or long-latency instruction, a field to designate that the instruction is a load instruction or a long-latency instruction, a field to indicate whether the load instruction is a pair instruction, and a field to indicate a load group (ldg) instruction. The E-stage entry also includes a field to indicate whether the destination register rd is 0x02 since a "CALL" instruction writes the return address into register r2. The register r2 is not specified explicitly in the instruction so that, for an unfinished load writing to r2, then the "CALL" instruction is to stall until the load finishes updating register r2. The E-stage entry also includes a field to indicate whether the E-stage entry is valid, which occurs in the E-stage. When the instruction transitions from the E-stage to the A1-stage, the E-stage entry becomes invalid and either the load entries or the long latency entry of the scoreboard is updated.

The number of entries allocated for loads is equal to the number of entries in the load buffer of the load/store unit **218**. The processor **100** allows one load hit under four misses requiring five load entries in the scoreboard. Fields in the load entries include an 8-bit register for holding the destination address (rd) specifier of

the load instruction, a field to indicate a group load (ldg), a field to indicate a load pair or load long, a field stage representing the age in the scoreboard of the instruction, a group load count indicating the number of finished loads in a group load, a field to indicate whether the entry is valid, and a field to indicate whether the load is to register r2. Stage bits are shifted right by one position each cycle. If a trap is detected before the load reaches write-back stage, the entry is invalidated.

If a long latency operation is not finished before a more recently issued long latency operation is received in the general functional unit **222**, the pipeline control unit **226** stalls the pipeline in the decoding (D) stage.

Instructions from all functional units (GFU, MFU1, MFU2, and MFU3) access the scoreboard in the decode (D) stage to check for dependencies. Usage of a single centralized scoreboard structure would require all source and destination register specifiers of all instructions to be routed to the scoreboard, increasing routing congestion and degrading scoreboard access time. Therefore, the pipeline control unit **226** replicates the scoreboard for each functional unit, but with all pointers to write and update or reset the entries in all scoreboards generated in a single control block within the pipeline control unit **226** and routed to all scoreboards.

When a pair instruction enters the decode (D) stage of a media functional unit **220**, the data and output dependencies of the inherent second instruction of the pair are also checked in the same cycles. Accordingly, seven read ports are used for the scoreboards of one media functional unit **220** including three source operands and one destination operand for the first instruction and two source operands and one destination operand for the inherent second instruction of the pair.

Instructions executed in the general functional unit **222** use four read ports.

When a load returns from either the load/store unit **218** or data cache **106**, the valid signal is asserted later in the cycle so that insufficient time remains in the cycle to generate reset pointers to invalidate the corresponding entry in the scoreboard. If a load returns in cycle N, the entry to be invalidated or updated is computed in cycle N+1 and the updated valid or ldg count bits are visible in cycle N+2.

A load enters the scoreboard at the first invalid entry corresponding to loads.

When the load buffer becomes full, the load/store unit **218** asserts a buffer full flag in E-stage of the last load which filled the load buffer. The signal becomes late in the cycle due to the transfer time in the round trip path between the pipeline control unit **226** and the load/store unit **218**. The pipeline control unit **226** latches the buffer full flag and stalls any load or setir instruction in the E-stage. The load/store unit **218** uses the load buffer to stage the data written to internal registers by the setir instruction.

When the load enters the scoreboard, the encoded index is sent to the load/store unit **218**. When the

load/store unit **218** returns the load, the load/store unit **218** also sends back the index to the scoreboard. The entry corresponding to the index is accessed to reset the scoreboard. The index-based scoreboard facilitates resetting of the scoreboard by avoiding comparison of the load_rd address with all the rd specifiers in the scoreboard to find the entry which is to be reset or updated. A load can be entering one entry in the scoreboard while another load is updating or invalidating another entry in the same cycle.

A load returning from the load/store unit **218** resets the scoreboard for a normal load or a ldpair or ldg which is updating the last pair of registers. For the ldg instruction which occurs during updating of the registers, only the group load count is changed.

The entry corresponding to a long latency operation is reset when the long latency operation enters A4 or T stage without being stalled.

Instructions access the scoreboard in D-stage with the rd specifier of the load returned in the previous cycle compared with the rs/rd specifiers of the current instruction to determine whether the load has returned. A dependency occurs only if a scoreboard match occurs and the data does not return from the load/store unit **218** or data cache **106** in the previous cycle.

The scoreboard is checked for data dependency and output dependency. For a load group instruction, only part of the source specifier bit are compared and the compare signal is disqualified if the load group count bits indicate the source specifier has already returned. For a load pair, part of the bits of the source specifier are compared to the destination specifier held in the scoreboard. For other instructions, all bits of the source specifier are compared to the destination specifier in the scoreboard.

For long latency instructions, data dependency is checked by comparing the destination specifier of the long entry against the source specifier. For output dependency of all instructions other than gfu load group, load pair, and call instructions, all bits of the long entry destination specifier are compared against the source specifier of the instruction. For the gfu load group, load pair, and call instructions, only part of the bits of the long entry destination specifier are compared against the source specifier of the instruction.

All matches from long entry are disqualified for gfu and mfu1 instructions if the long latency entry is in a final execute stage in the cycle since gfu and mfu1 instructions can bypass the results of long latency operations from the final execute stages.

Results of long latency operations enter an A4-stage register in the annexes. Instructions executed in mfu2 and mfu3 can bypass results held in the A4-stage register.

Several definitions are set forth for describing stalls. The term "matching a load entry" refers to a scoreboard hit in which the data has not yet returned from the data cache **106** or the load/store unit **218**. Source

specifiers for the media functional units (mfu) **220** are rs1, rs2, and rs3, except for pair instructions in which the source specifiers are rs1, rs2, rs1h, and rs2h. Source specifiers for the general functional unit (gfu) **222**, except for branches and conditional moves, are rs1 and rs2. Source specifiers for stores are rs1, rs2, and rd. Source specifiers for branches are rd. Source specifiers for conditional move (cmove) instructions are rs1, rs2, and rd.

Decoder stage (D-stage) stalls for instructions executed in the general functional unit **222** occur under several conditions including the occurrence of an output dependency when a previous load is unfinished. A D-stage stall occurs when an output dependency occurs with a long-latency entry when the long-latency entry is not in the final execute stage of the cycle. A D-stage stall occurs when the rd/rs specifier for the instruction matches a scoreboard entry, a condition termed a "load use immediate/ long use immediate case".

A D-stage stall also occurs when the instruction in the D-stage is a long-latency instruction and a valid long-latency entry already exists in the scoreboard where the long-latency entry in the scoreboard does not reset the scoreboard in that cycle either because the instruction associated with the long-latency entry has not finished or is finished but held in a hold register due to a port conflict.

A D-stage stall also results when the processor **100** is in a step mode that is part of a debug functionality. In the step mode, all instructions in one VLIW group stall in the D-stage until the pipelines are empty. Following the stall, data for the source operands is received from the register file segments **224** and execution proceeds.

D-stage stalls for media functional unit **220** instructions of mfu1 are the same as D-stage stalls for general functional unit **222** instructions except that mfu1 instructions stall in the D-stage for all data dependencies and output dependencies of the inherent second instruction of a pair. Media functional unit **220** instructions of mfu2 and mfu3 stall in the D-stage for any source/destination specifier match in the scoreboard.

For various instructions, generation of a D-stage stall includes accessing the scoreboard for a load or long-latency dependency for all rs/rd specifiers, then qualifying scoreboard matches and ORing the scoreboard matches. The time for D-stage stall generation lasts for more than half the clock cycle.

To preserve the VLIW nature of instructions, even though each instruction in a VLIW group may enter the E-stage at different cycles, all instructions in the group wait in the E-stage until the full VLIW group has entered the E-stage. Accordingly, the instructions stall in the E-stage until the entire VLIW group is available for execution. Each instruction stalls in the E-stage for a number of cycles that is generally independent of the operation of other individual instructions within the group. When an instruction is waiting in the E-stage for other instructions in the group, the waiting instruction generates a local E-stage stall signal (gf_local_stalle or mf_local_stall).

In addition to the E-stage stall, instructions executed in the general functional unit **222** also stall for the cases of (i) a load use dependency with a previous load, (ii) a gfu instruction with load or store buffer full, (iii) a

stall for synchronizing long latency pair gfu instructions, (iv) stalls to avoid port conflicts, (v) stalls to handle load-use dependencies, (vi) a "membar" instruction, and (vii) a processor in a step mode of the load/store unit **218**. The membar instruction is a memory access instruction that specifies that all memory reference instructions that are already issued must be performed before any subsequent memory reference instructions may be initiated.

For the load use dependency with a previous load, if a gfu or mfu1 instruction enters the E-stage with a load dependency, the instruction does not assert the load dependency E-stage stall until all other sub instructions of the VLIW group enter the E-stage, but the instruction does bypass the load data returning from the load/store unit **218**.

Another stall condition occurs when the general functional unit **222** executes an instruction such as load, setir, prefetch, or cas, and the load buffer is full. Alternatively, the general functional unit **222** executes a store or cas instruction and the store buffer is full, in which case the gfu instruction also generates an E-stage stall. The load/store unit **218** sends a "pcu-lsu loadbuffer full" signal and a "lsu-pcu store buffer full" signal to the pipeline control unit **226**. The pipeline control unit **226** latches the signals and uses the signals during the E-stage to generate stalls.

An instruction executed by the general functional unit **222** stalls in E-stage for either one cycle or three cycles so that the instructions reach the T-stage simultaneously with five cycle and seven cycle latency pair instructions in the same VLIW group.

E-stage stalls are also invoked to avoid port conflicts. If a group N+1 includes a valid instruction in a position of the media functional units **220** for which the instruction is a 7 cycle or 5 cycle pair instruction in the group N, then all instructions in the group N+1 stall for an additional cycle in the E-stage so that the mfu instruction in the group N+1 does not finish in the same cycle as the inherent second instruction of the pair. The stall signal generated is a "previous pair stall" (gf_prev_pair_stall_e).

An E-stage stall is generated when the instruction in the first position of the media functional units **220** (mfu1) has a load-use dependency with a previous unfinished load instruction. The mfu1 instruction asserts the stall (mf1_ld_stalle) only after all the instructions in the VLIW group reach the E-stage.

While the invention has been described with reference to various embodiments, it will be understood that these embodiments are illustrative and that the scope of the invention is not limited to them. A typical implementation will generally include two, three, four, or more processors, although any suitable number of processors may be included.

Many variations, modifications, additions and improvements of the embodiments described are possible. For example, those skilled in the art will readily implement the steps necessary to provide the structures and methods disclosed herein, and will understand that the process parameters, materials, and dimensions are given by

way of example only and can be varied to achieve the desired structure as well as modifications which are within the scope of the invention. Variations and modifications of the embodiments disclosed herein may be made based on the description set forth herein, without departing from the scope and spirit of the invention as set forth in the following claims. For example, the described structure and operating method may be applied to a particular functional unit/ register file segment pair cluster structure including a set number and type of functional units and divided register file structure, other suitable structures may otherwise be employed. The number of functional unit/ register file segment pair clusters may be varied, and other types of functional units employed. The register file may be replaced by any type of executional storage for storing results generated by the functional units. The register file and processor may be any suitable size. The register file may be much larger or much smaller than the described 128 registers. The very long instruction word may include any suitable number of subinstructions.

WE CLAIM

1. A processor comprising:

a plurality of parallel execution paths that selectively execute instructions in a plurality of operating modes including:

an extended-width operating mode in which instructions execute in coordination across the parallel execution paths; and

a multiple-thread operating mode in which instructions execute in parallel across a plurality of independent threads, the execution paths including a multiple instruction parallel pathway within a thread.

2. A processor according to Claim 1 wherein:

the independent parallel execution paths include functional units that execute an instruction set including special data handling instructions supporting a multiple-thread execution environment.

3. A processor according to Claim 1 wherein:

the processor is a Very Long Instruction Word (VLIW) processor.

4. A processor according to Claim 1 further comprising:

control logic that controls a plurality of processing units in the parallel execution paths to execute instructions mutually independently in a plurality of independent execution threads.

5. A processor according to Claim 1 further comprising:

control logic that controls a plurality of processing units in the parallel execution paths with a low thread synchronization to operate in combination using spatial software pipelining in the manner of a single wide-issue processor.

6. A processor according to Claim 1 further comprising:

control logic that controls a plurality of processing units in the parallel execution paths in a multiple-thread operation on the basis of a highly parallel structure including multiple independent parallel execution paths for executing in parallel across threads and a multiple-instruction parallel pathway within a thread.

7. A processor according to Claim 1 further comprising:

control logic that controls a plurality of processing units in the parallel execution paths in a single-thread

wide-issue operation on the basis of the highly parallel structure including multiple parallel execution paths with low level synchronization for executing the single wide-issue thread.

8. A processor according to Claim 1 wherein:
the plurality of independent parallel instruction paths selectively execute as a plurality of processors in multiple-threaded applications using a Java™ programming language running under a multiple-threaded operating system on a multiple-threaded Java Virtual Machine™.

9. A processor according to Claim 1 wherein:
the processor includes two independent processor elements forming a respective two independent parallel execution paths.

10. A processor according to Claim 1 wherein:
in a first selected mode of operation the plurality of independent parallel instruction paths execute as a plurality of processors in multiple-threaded applications using a Java™ programming language that generates a plurality of threads that respectively execute in the plurality of independent parallel instruction paths with a minimum of threading overhead; and
in a second selected mode of operation the plurality of independent parallel instruction paths execute as a single processor in extended-width applications using the Java™ programming language that generates an extended-width thread.

11. A processor according to Claim 1 wherein:
the independent processor elements are integrated into a single integrated-circuit chip.

12. A processor comprising:
a plurality of processor elements in a single integrated circuit chip capable of executing a respective plurality of threads concurrently during a multiple-threaded operation; and
a control logic that selectively controls an execution environment in which the plurality of processor elements execute multiple independent execution threads execute simultaneously, and alternatively controls the plurality of processor elements execute in the parallel execution paths with a low thread synchronization to operate in combination using spatial software pipelining in the manner of a single wide-issue processor.

13. A processor according to Claim 12 wherein:
the processor elements are Very Long Instruction Word (VLIW) processors forming a respective plurality of independent parallel execution paths.

14. A processor according to Claim 12 wherein:
the processor is a general-purpose processor.

15. A processor according to Claim 12 wherein:
the processor includes two independent processor elements in a single integrated circuit chip.

16. A processor according to Claim 12 wherein:
the independent processor elements include a plurality of functional units that execute a respective plurality of instructions concurrently and in parallel.

17. A processor according to Claim 12 wherein:
a plurality of independent processor elements are Very Long Instruction Word (VLIW) processor elements that include a plurality of functional units operating concurrently in parallel, the functional units including media functional units operating as digital signal processors, and a general functional unit, and
the media functional units capable of executing a instruction that executes both a multiply operation and an addition operation in a single cycle, the multiply operation and add operations being either floating point or fixed point.

18. A processor according to Claim 12 wherein:
the control logic controls a plurality of processing units in the parallel execution paths to execute instructions mutually independently in a plurality of independent execution threads.

19. A processor according to Claim 12 wherein:
the control logic controls a plurality of processing units in the parallel execution paths with a low thread synchronization to operate in combination using spatial software pipelining in the manner of a single wide-issue processor.

20. A processor comprising:
a plurality of processor elements in a single concurrently executable parallel processor, the processor elements including:
an instruction supply logic;
an instruction preparation logic coupled to the instruction supply logic;
a plurality of functional units coupled to the instruction supply logic and coupled to the instruction preparation logic;

a register file coupled to the plurality of functional units, coupled to the instruction supply logic, and coupled to the instruction preparation logic,

the instruction supply logic, the instruction preparation logic, the plurality of functional units, and the register file for a first independent processor element being independent and separate from the instruction supply logic, the instruction preparation logic, the plurality of functional units, and the register file of a second independent processor element;

a control logic that selectively controls an execution environment in which the plurality of processor elements execute multiple independent execution threads execute simultaneously, and alternatively controls the plurality of processor elements execute in the parallel execution paths with a low thread synchronization to operate in combination using spatial software pipelining in the manner of a single wide-issue processor; and

a data cache coupled to and shared among the plurality of independent processor elements.

21. A processor according to Claim 20 wherein:

the plurality of independent processor elements are capable of executing a respective plurality of threads concurrently during a multiple-threaded operation and capable of executing in combination in a single extended-width thread in parallel.

22. A processor according to Claim 20 wherein:

the plurality of independent processor elements are integrated into a single integrated-circuit chip.

23. A processor according to Claim 20 wherein:

an instruction supply logic includes an instruction cache for a first independent processor element that is independent and separate from an instruction cache of the instruction supply logic of a second independent processor element.

24. A processor according to Claim 20 wherein:

the control logic controls the processor for applications with a high level of instruction-level parallelism to execute a plurality of instructions in parallel on the plurality of processor elements using a low thread synchronization overhead to operate with a level of performance of an increased-width VLIW processor; and

the control logic controls the processor for applications with a low level of instruction-level parallelism to execute a plurality of independent threads in parallel on the plurality of processor elements.

25. A processor according to Claim 20 wherein:

the processor is a Very Long Instruction Word (VLIW) processor including two independent four-wide VLIW processor elements and the control logic controls the processor to selectively execute in a first mode as an eight-wide VLIW processor and to alternatively execute in a second mode as two four-wide VLIW threads.

26. A processor comprising:

a plurality of selectively independent parallel instruction paths executing instructions in parallel across threads and a multiple-instruction pathway within a thread, and alternatively executing the parallel instruction paths in combination in a single thread;

low overhead interconnections between the parallel instruction paths to selectively operate the plurality of parallel instruction paths either in coordination as a combined-width VLIW processor with subinstructions in the multiple independent parallel execution paths extending across the execution paths, or independently as a plurality of threads.

27. A processor according to Claim 26 further comprising:

a plurality of functional units that execute an instruction set including special data handling instructions that are useful in a multiple-thread environment.

28. A general-purpose processor comprising:

a plurality of processor elements in a single integrated circuit die, the plurality of processor elements executing selectively in multiple modes including:

a multiple-threaded mode in which multiple processor elements execute concurrently in multiple-threaded operation; and

an extended-width instruction mode in which multiple processor elements execute concurrently in coordination in a single thread across a plurality of functional units.

29. A processor according to Claim 28 wherein:

the processor is a VLIW processor including a plurality of VLIW processor elements, the VLIW processor elements including a plurality of functional units that execute in parallel.

30. A processor according to Claim 28 wherein:

the processor is a VLIW processor including a two VLIW processor elements, the VLIW processor elements including four functional units that execute in parallel, the four functional units including three media functional units and one general functional unit.

31. A processor according to Claim 28 wherein:

the processor is a VLIW processor including a two VLIW processor elements, the VLIW processor elements including four functional units that execute in parallel, the four functional units including three media functional units and one general functional unit, the processor executing in an extended-width mode in combination on the eight functional units, and the processor executing in a multiple-thread mode in parallel in two threads, one thread executing on each of the two VLIW processor elements.

32. A method of operating a processor comprising:

selecting an execution mode between a multiple-thread mode and an extended-width instruction mode;

selectively executing instructions in a plurality of parallel execution paths including:

executing the instructions in the multiple-thread mode in parallel across a plurality of independent threads, the execution paths including a multiple instruction parallel pathway within a thread; and

executing the instructions in the extended-width instruction mode in coordination across the parallel execution paths.

33. A method according to Claim 32 further comprising:

executing an instruction set including special data handling instructions supporting a multiple-thread execution environment.

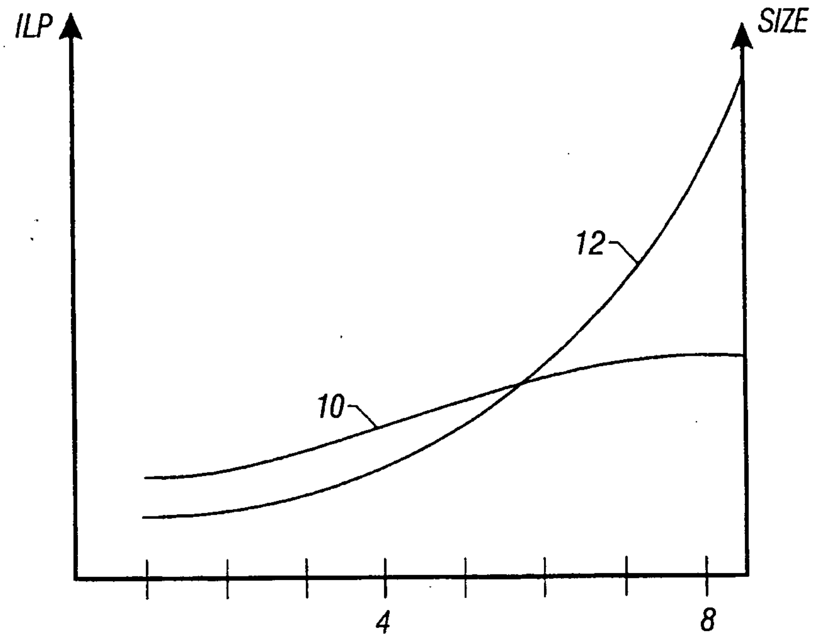



FIG. 1A

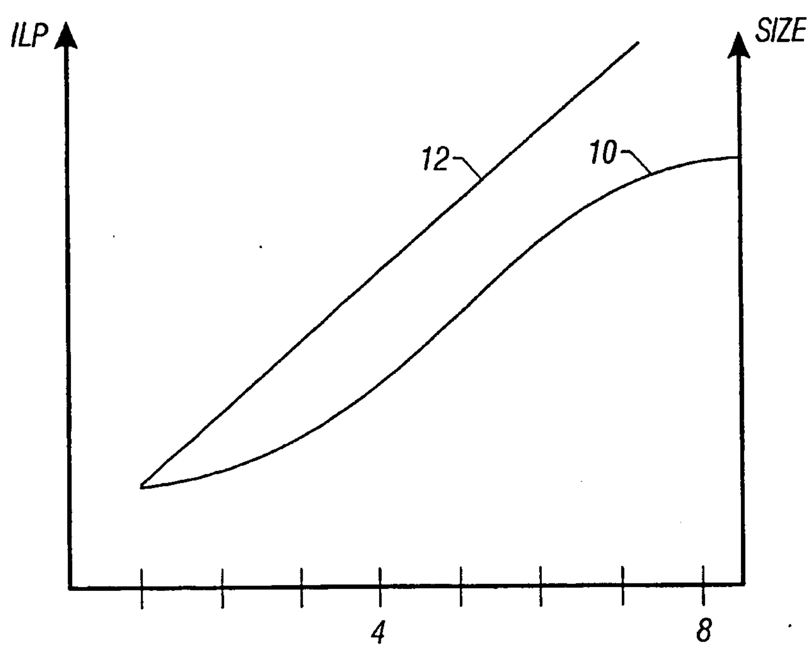



FIG. 1B

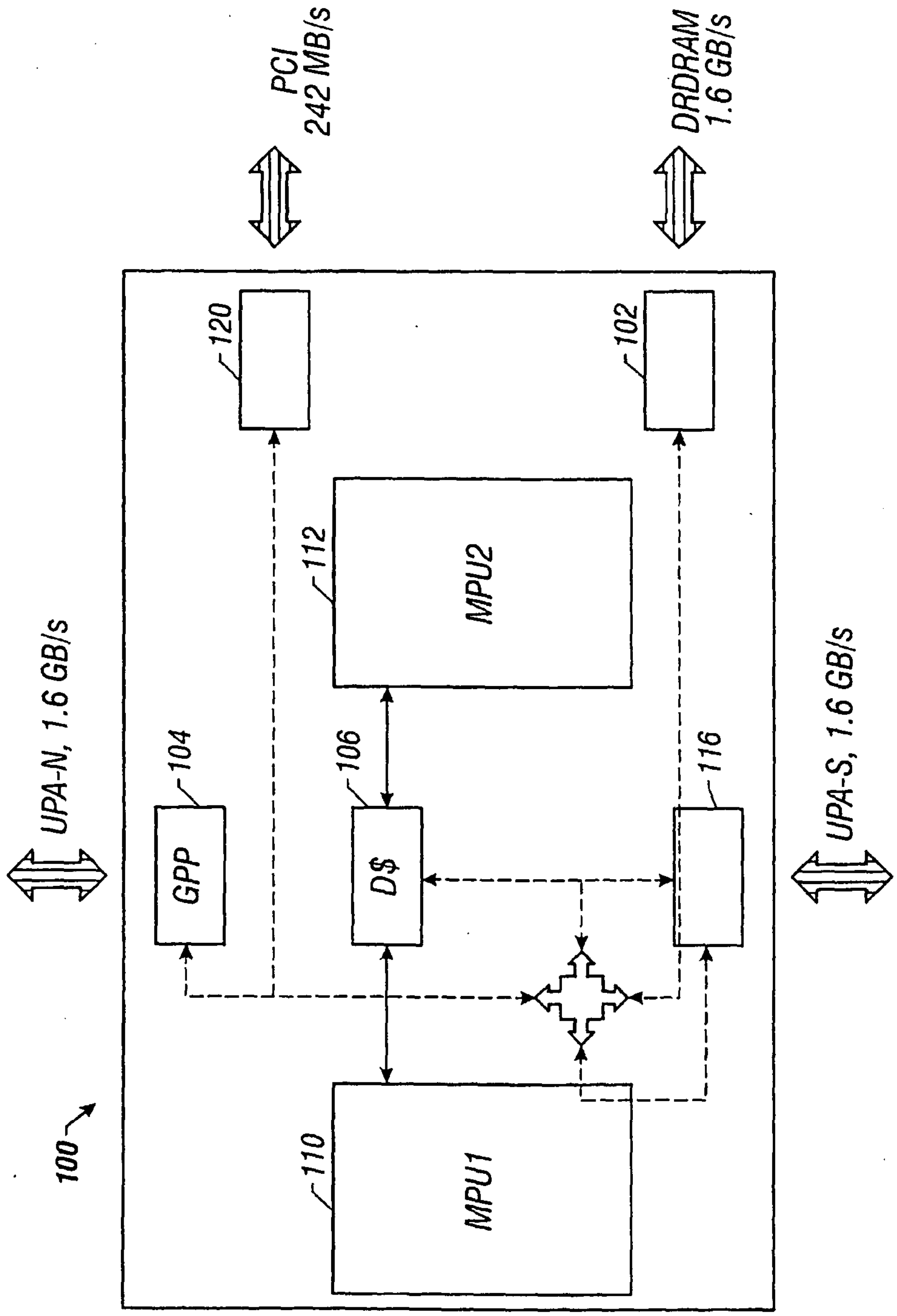

PCI
242 MB/s
DRDRAM
1.6 GB/s
UPA-N, 1.6 GB/s
UPA-S, 1.6 GB/s
120
102
112
MPU2
104
GPP
106
D$
116
110
MPU1
100

FIG. 2

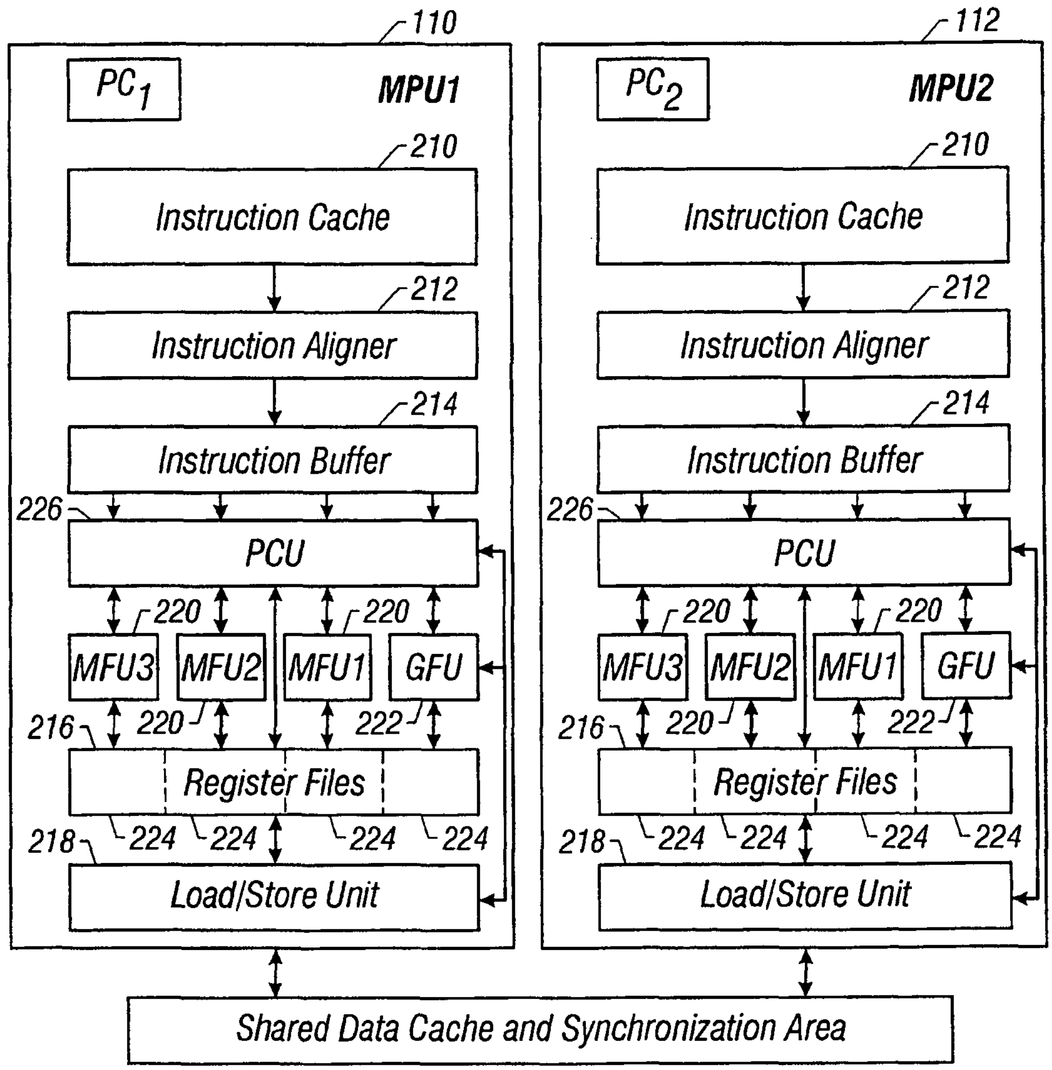

110
112
$PC_1$
MPU1
$PC_2$
MPU2
210
Instruction Cache
212
Instruction Aligner
214
Instruction Buffer
226
PCU
220
MFU3
MFU2
MFU1
GFU
216
222
Register Files
218
224
Load/Store Unit
Shared Data Cache and Synchronization Area

FIG. 3

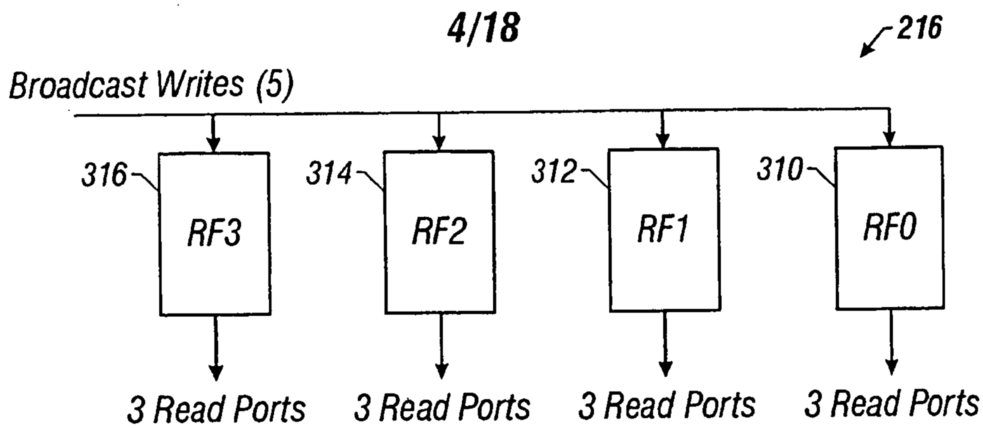



FIG. 4

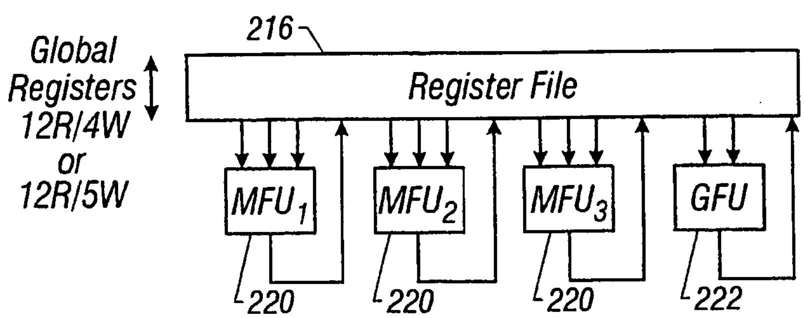



FIG. 5A

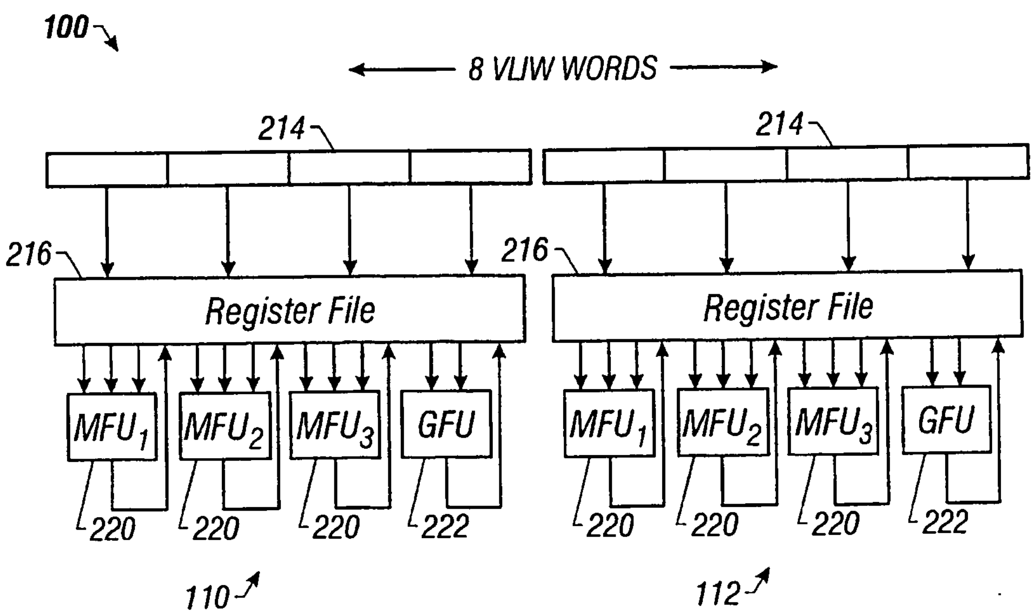



FIG. 5B

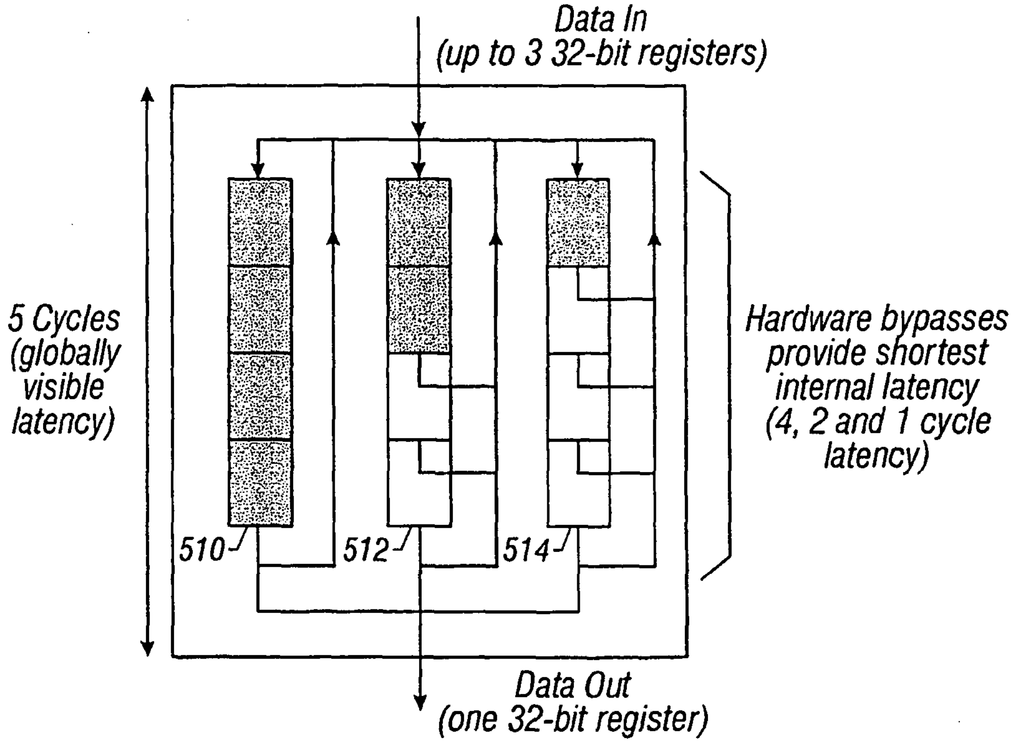

Data In
(up to 3 32-bit registers)
5 Cycles
(globally
visible
latency)
Hardware bypasses
provide shortest
internal latency
(4, 2 and 1 cycle
latency)
510
512
514
Data Out
(one 32-bit register)

FIG. 6

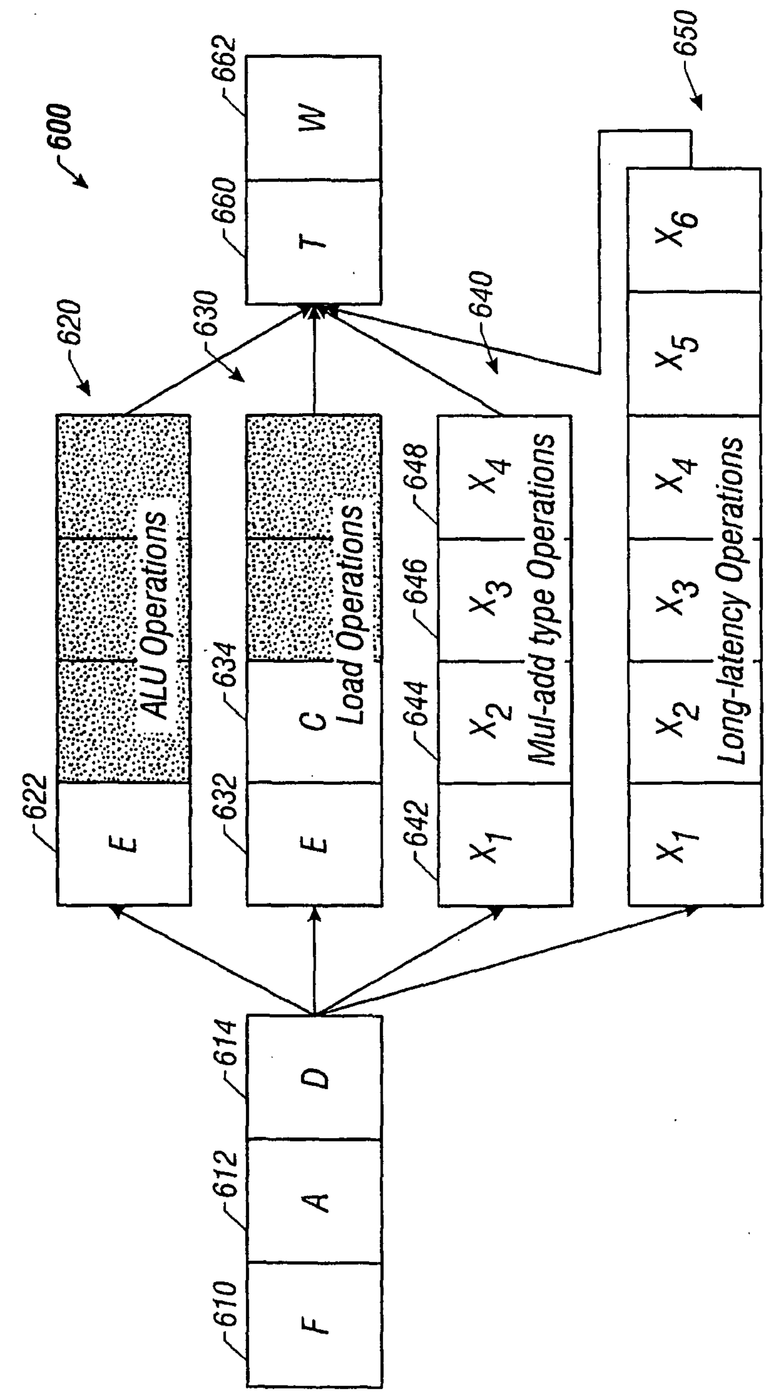
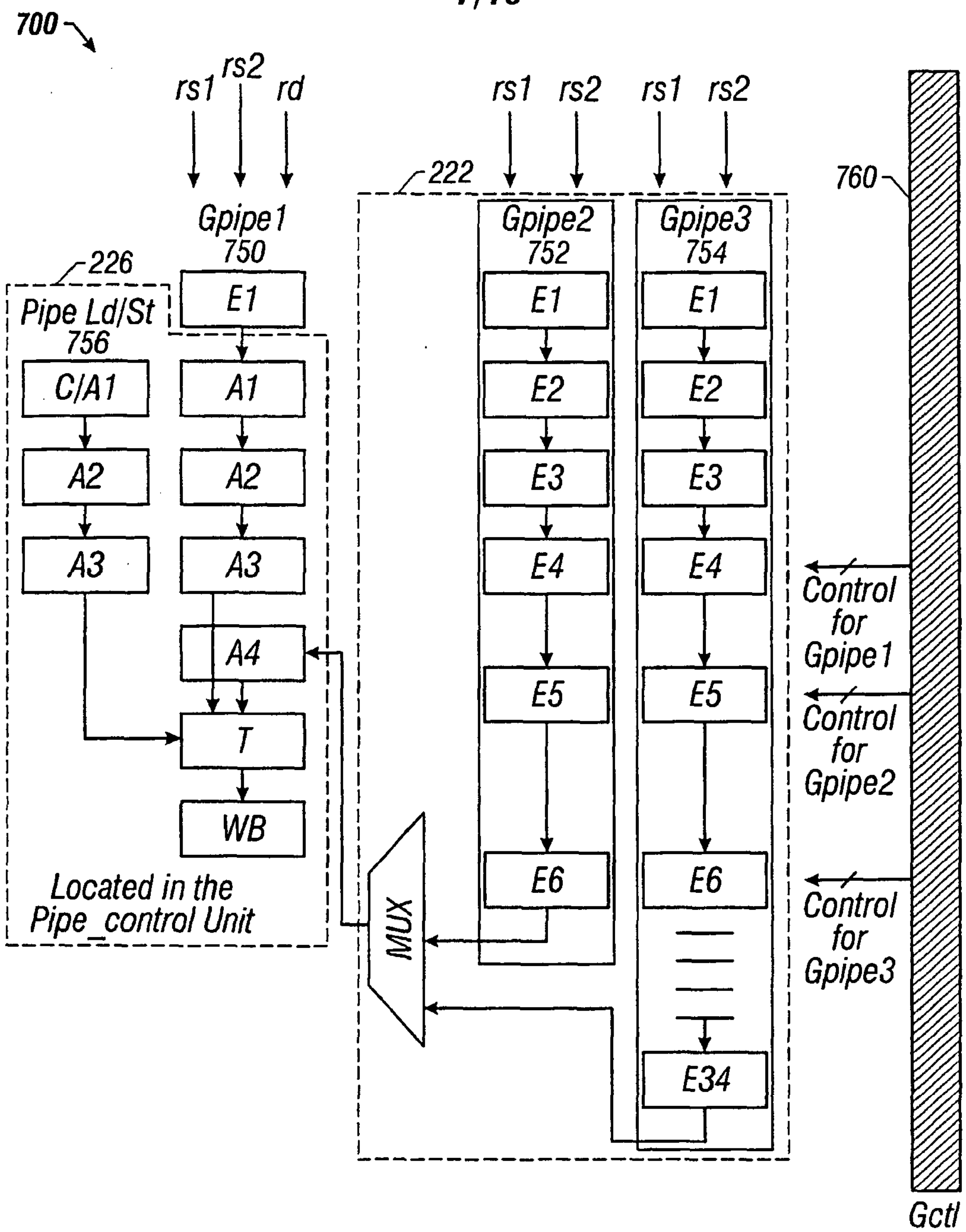

600
620
630
640
650
622
632
634
642
644
646
648
660
662
610
612
614
F
A
D
E
E
C
T
W
$X_1$
$X_2$
$X_3$
$X_4$
$X_5$
$X_6$
ALU Operations
Load Operations
Mul-add type Operations
Long-latency Operations

FIG. 7



700
rs1
rs2
rd
rs1
rs2
rs1
rs2
222
760
226
Gpipe1
750
Gpipe2
752
Gpipe3
754
Pipe Ld/St
756
E1
C/A1
A1
A2
A3
A4
T
WB
E2
E3
E4
E5
E6
E34
MUX
Control for Gpipe1
Control for Gpipe2
Control for Gpipe3
Located in the Pipe_control Unit
Gctl


FIG. 8A

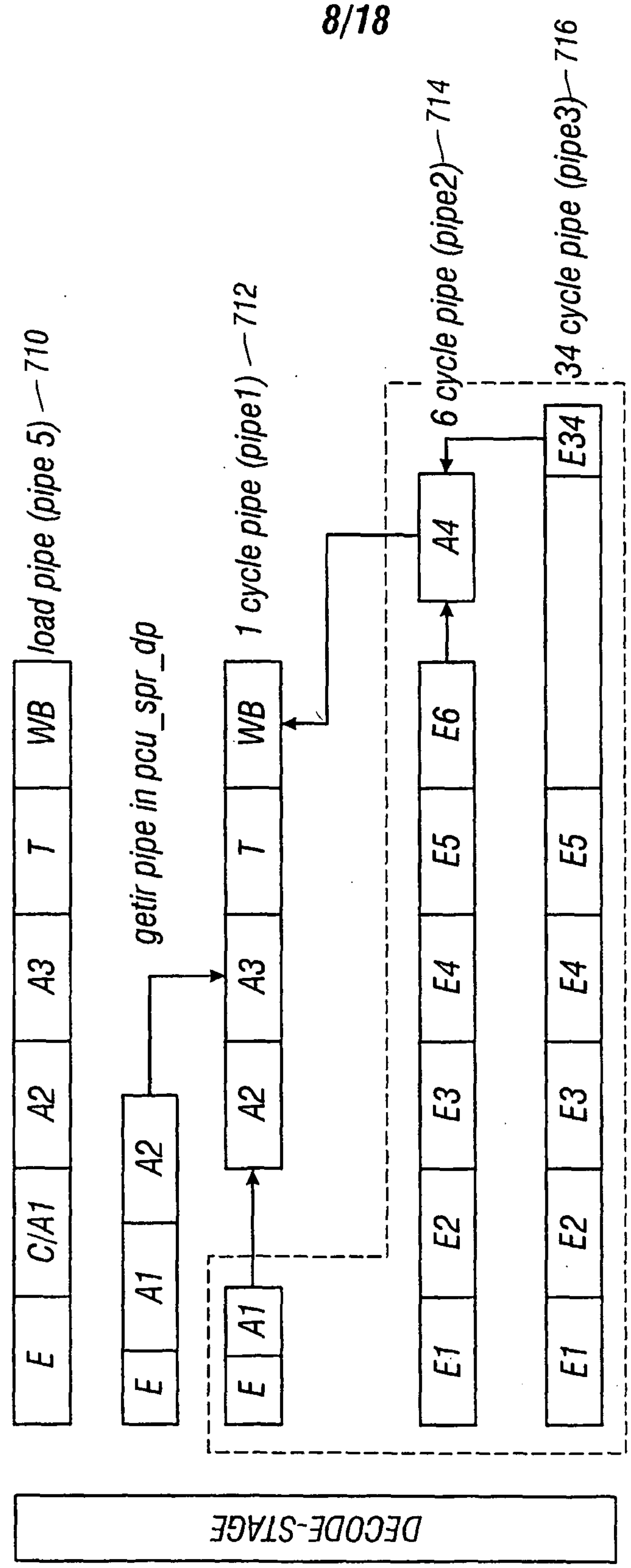

load pipe (pipe 5)
710
E
C/A1
A2
A3
T
WB
getir pipe in pcu_spr_dp
E
A1
A2
1 cycle pipe (pipe1)
712
E
A1
A2
A3
T
WB
6 cycle pipe (pipe2)
714
E1
E2
E3
E4
E5
E6
A4
34 cycle pipe (pipe3)
716
E1
E2
E3
E4
E5
E34
DECODE-STAGE

FIG. 8B

700

| | | | | | | |
|---|---|---|---|---|---|---|
| level_1 | gfu<br>E | ldx<br>ldx1 | mfu1<br>E | mfu2 | mfu3 | long<br>E6E34 |
| level_2 | A1 | ldx2 | A1/E2 | | | A4 |
| level_3 | gfu<br>A2 | ldx<br>ldx3 | mfu1<br>A2 | mfu2 | mfu3 | long |
| level_4 | A3 | ldx4 | A3/E4 | | | |
| level_5 | T | | T | T | T | |
| level_6 | WB | | WB | WB | WB | |
| level_7 | RFO | | | | | |

FIG. 8C

800

| | | | | | | |
|---|---|---|---|---|---|---|
| E | A1 | A2 | A3 | T | WB | 810 |
| E1 | E2 | A2 | A3 | T | WB | 812 |
| E1 | E2 | E3 | E4 | T | WB | 814 |

FIG. 9

| group | mfu3 | mfu2 | mfu1 | gfu |
|---|---|---|---|---|
| vliw_1 | mfu3_1 | mfu2_1 | mfu1_1 | gfu_1 |
| vliw_2 | mfu3_2 | mfu2_2 | mfu1_2 | gfu_2 |
| vliw_3 | mfu3_3 | mfu2_3 | mfu1_3 | gfu_3 |
| vliw_4 | mfu3_4 | mfu2_4 | mfu1_4 | gfu_4 |
| vliw_5 | mfu3_5 | mfu2_5 | mfu1_5 | gfu_5 |

FIG. 10A

| cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| mfu1_1 | D | E1 | E2 | X | E3 | E4 | T | WB | | | | | |
| helper | | D | E1 | E2 | X | E3 | E4 | T | WB | | | | |
| gfu_1 | D | E | E | A1 | A2 | A3 | T | WB | | | | | |
| gfu_2 | | D | D | E | E | A1 | A2 | A3 | T | WB | | | |

FIG. 10B

| cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| mfu_1 | D | E1 | E2 | X1 | X2 | X3 | E3 | E4 | T | WB | | | |
| helper | | D | E1 | E2 | X1 | X2 | X3 | E3 | E4 | T | WB | | |
| gfu_1 | D | E | E | E | E | A1 | A2 | A3 | T | WB | | | |
| vliw_2 | | D | D | D | D | E | E | A1 | A2 | A3 | T | WB | |

FIG. 10C

| cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| mfu1_1 | D | E1 | E2 | X1 | X2 | X3 | E3 | E4 | T | WB | | |
| helper | | D | E1 | E2 | X1 | X2 | X3 | E3 | E4 | T | WB | |
| mfu1_2 | | | D | D | D | E | E | A1 | A2 | A3 | T | WB |
| | | | | | | | | | | | | |
| mfu2_1 | D | E | E | E | E | A1 | A2 | A3 | T | WB | | |
| mfu2_2 | | | D | D | D | E1 | E2 | X | E3 | E4 | T | |
| helper | | | | | | D | E1 | E2 | X | E3 | E4 | T |
| | | | | | | | | | | | | |
| gfu_1 | D | E | E | E | E | A1 | A2 | A3 | T | WB | | |
| gfu_2 | | D | D | D | D | E | E | A1 | A2 | A3 | T | WB |

*FIG. 11A*

| cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| mfu1_1 | D | E1 | E1 | E1 | E1 | E2 | A2 | A3 | T | WB | | |
| helper | | D | D | D | D | E1 | E2 | A2 | A3 | T | WB | |
| mfu1_2 | | | | | | D | E | A1 | A2 | A3 | T | WB |
| | | | | | | | | | | | | |
| mfu2_1 | D | E1 | E1 | E1 | E2 | X | E3 | E4 | T | WB | | |
| helper | | D | D | D | E1 | E2 | X | E3 | E4 | T | WB | |
| mfu2_2 | | | | | D | D | E | A1 | A2 | A3 | T | WB |
| | | | | | | | | | | | | |
| mfu3_1 | D | E1 | E2 | X1 | X2 | X3 | E3 | E4 | T | WB | | |
| helper | | D | E1 | E2 | X1 | X2 | X3 | E3 | E4 | T | WB | |
| mfu3_2 | | | D | D | D | D | E | A1 | A2 | A3 | T | WB |
| | | | | | | | | | | | | |
| gfu_1 | D | E | E | E | E | A1 | A2 | A3 | T | WB | | |
| gfu_2 | | D | D | D | D | D | E | A1 | A2 | A3 | T | WB |

FIG. 11B

| cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| mfu1_1 | D | E1 | E1 | E1 | E1 | E2 | A2 | A3 | T | WB | | |
| mfu1_2 | | D | D | D | D | E | E | A1 | A2 | A3 | T | WB |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| mfu2_1 | D | E1 | E1 | E1 | E2 | X | E3 | E4 | T | WB | | |
| helper | | D | D | D | E1 | E2 | X | E3 | E4 | T | WB | |
| mfu2_2 | | | | | D | E | E | A1 | A2 | A3 | T | WB |
| | | | | | | | | | | | | |
| mfu3_1 | D | E1 | E2 | X1 | X2 | X3 | E3 | E4 | T | WB | | |
| helper | | D | E1 | E2 | X1 | X2 | X3 | E3 | E4 | T | WB | |
| mfu3_2 | | | D | D | D | E | E | A1 | A2 | A3 | T | WB |
| | | | | | | | | | | | | |
| gfu_1 | D | E | E | E | E | A1 | A2 | A3 | T | WB | | |
| gfu_2 | | D | D | D | D | E | E | A1 | A2 | A3 | T | WB |

FIG. 11C

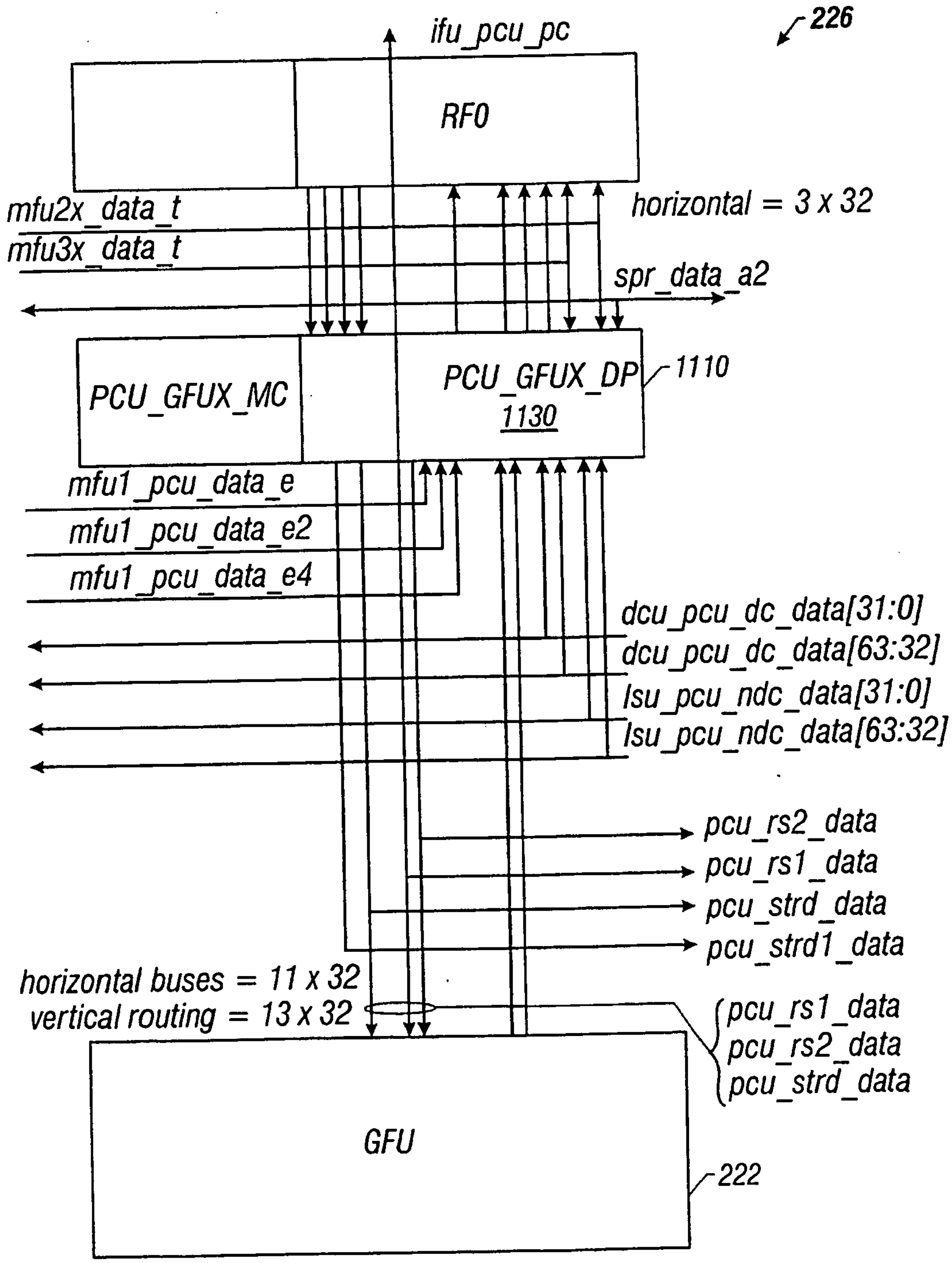

226
ifu_pcu_pc
RF0
horizontal = 3 x 32
mfu2x_data_t
mfu3x_data_t
spr_data_a2
PCU_GFUX_MC
PCU_GFUX_DP
1130
1110
mfu1_pcu_data_e
mfu1_pcu_data_e2
mfu1_pcu_data_e4
dcu_pcu_dc_data[31:0]
dcu_pcu_dc_data[63:32]
lsu_pcu_ndc_data[31:0]
lsu_pcu_ndc_data[63:32]
pcu_rs2_data
pcu_rs1_data
pcu_strd_data
pcu_strd1_data
horizontal buses = 11 x 32
vertical routing = 13 x 32
pcu_rs1_data
pcu_rs2_data
pcu_strd_data
GFU
222

FIG. 12A

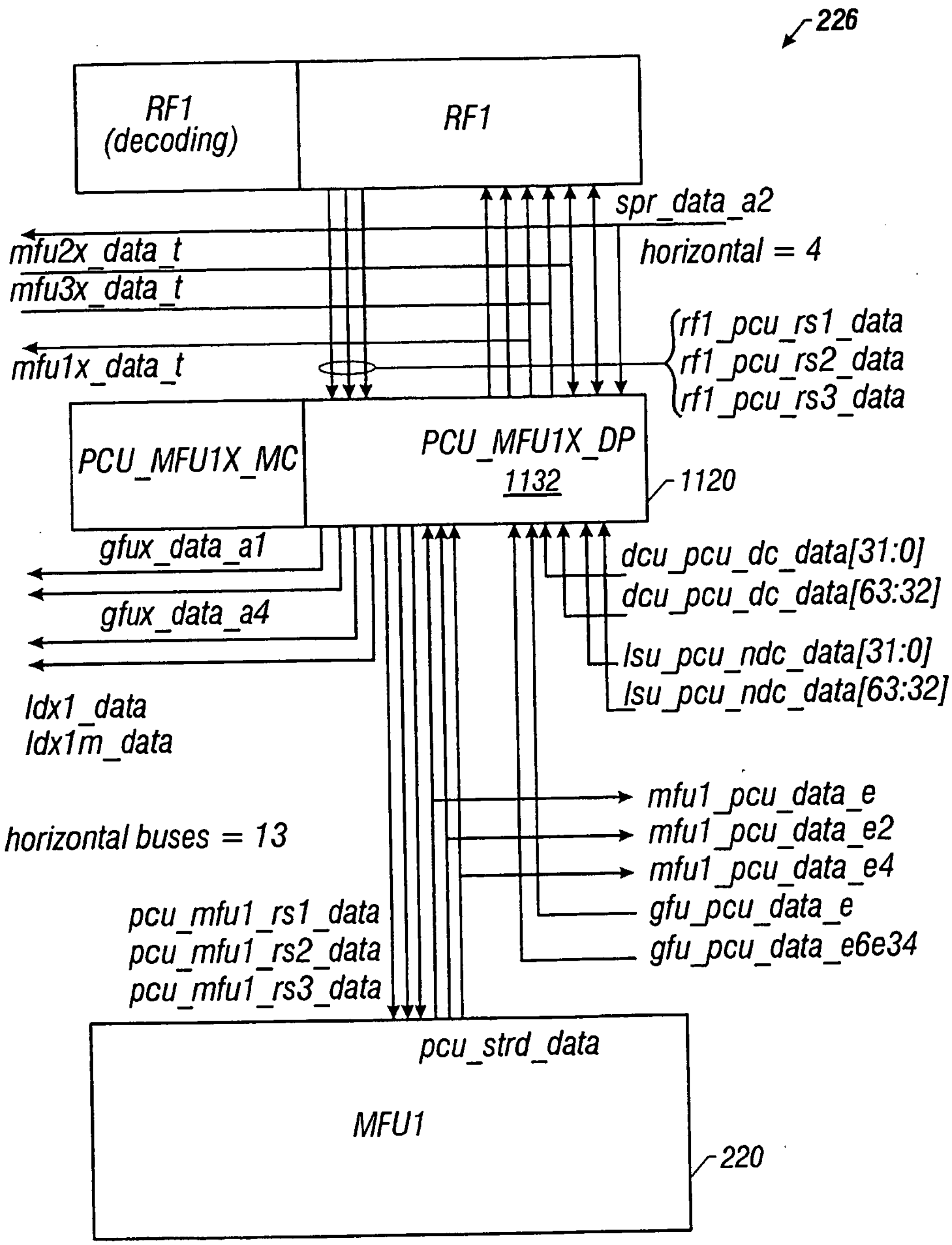

226
RF1 (decoding)
RF1
spr_data_a2
horizontal = 4
mfu2x_data_t
mfu3x_data_t
mfu1x_data_t
rf1_pcu_rs1_data
rf1_pcu_rs2_data
rf1_pcu_rs3_data
PCU_MFU1X_MC
PCU_MFU1X_DP
1132
1120
gfux_data_a1
gfux_data_a4
ldx1_data
ldx1m_data
dcu_pcu_dc_data[31:0]
dcu_pcu_dc_data[63:32]
lsu_pcu_ndc_data[31:0]
lsu_pcu_ndc_data[63:32]
horizontal buses = 13
mfu1_pcu_data_e
mfu1_pcu_data_e2
mfu1_pcu_data_e4
gfu_pcu_data_e
gfu_pcu_data_e6e34
pcu_mfu1_rs1_data
pcu_mfu1_rs2_data
pcu_mfu1_rs3_data
pcu_strd_data
MFU1
220

FIG. 12B

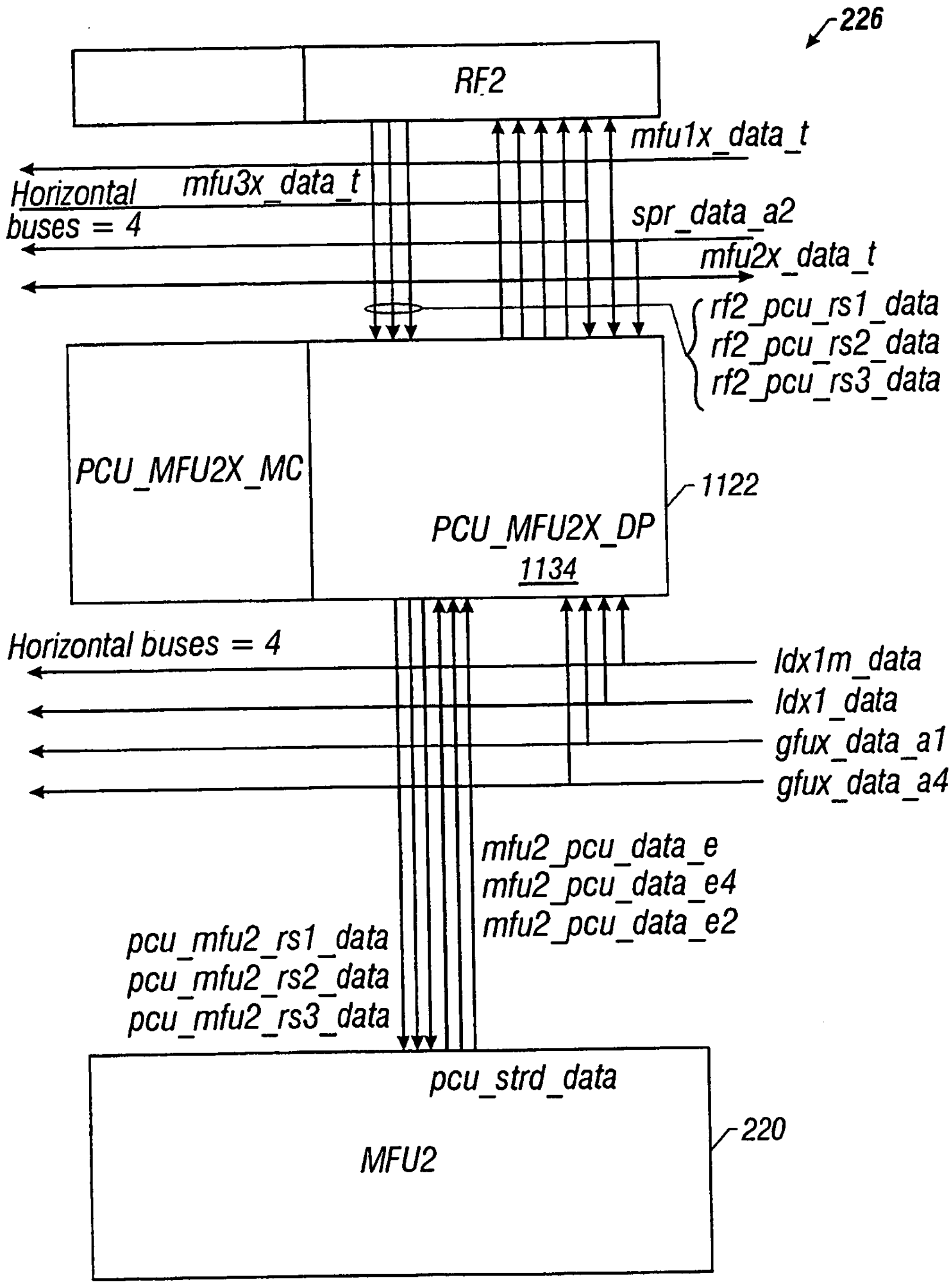

226
RF2
mfu1x_data_t
Horizontal buses = 4
mfu3x_data_t
spr_data_a2
mfu2x_data_t
rf2_pcu_rs1_data
rf2_pcu_rs2_data
rf2_pcu_rs3_data
PCU_MFU2X_MC
1122
PCU_MFU2X_DP
1134
Horizontal buses = 4
ldx1m_data
ldx1_data
gfux_data_a1
gfux_data_a4
mfu2_pcu_data_e
mfu2_pcu_data_e4
mfu2_pcu_data_e2
pcu_mfu2_rs1_data
pcu_mfu2_rs2_data
pcu_mfu2_rs3_data
pcu_strd_data
MFU2
220

FIG. 12C

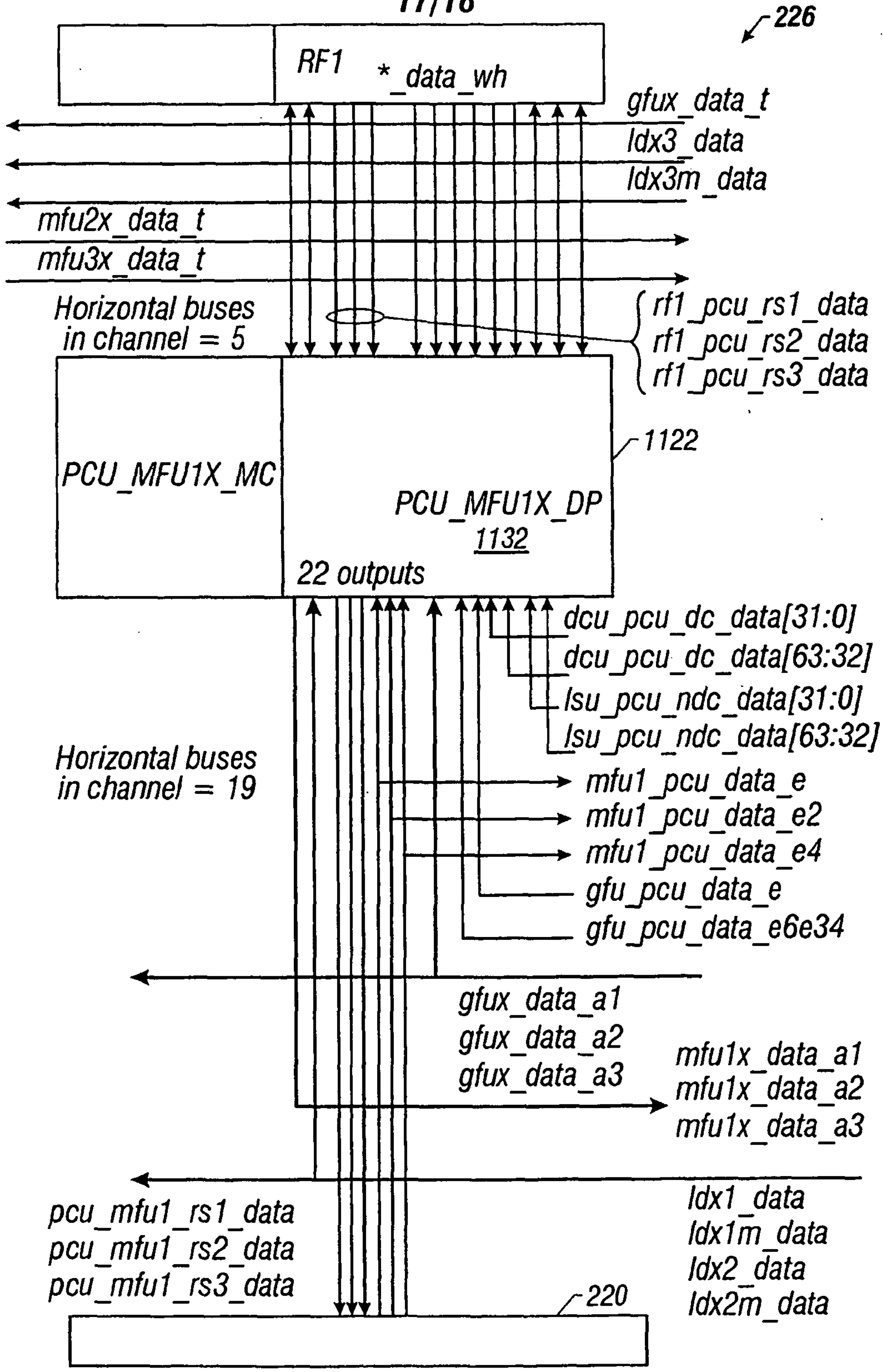

226
RF1
*_data_wh
gfux_data_t
ldx3_data
ldx3m_data
mfu2x_data_t
mfu3x_data_t
Horizontal buses in channel = 5
rf1_pcu_rs1_data
rf1_pcu_rs2_data
rf1_pcu_rs3_data
1122
PCU_MFU1X_MC
PCU_MFU1X_DP
1132
22 outputs
dcu_pcu_dc_data[31:0]
dcu_pcu_dc_data[63:32]
lsu_pcu_ndc_data[31:0]
lsu_pcu_ndc_data[63:32]
Horizontal buses in channel = 19
mfu1_pcu_data_e
mfu1_pcu_data_e2
mfu1_pcu_data_e4
gfu_pcu_data_e
gfu_pcu_data_e6e34
gfux_data_a1
gfux_data_a2
gfux_data_a3
mfu1x_data_a1
mfu1x_data_a2
mfu1x_data_a3
pcu_mfu1_rs1_data
pcu_mfu1_rs2_data
pcu_mfu1_rs3_data
ldx1_data
ldx1m_data
ldx2_data
ldx2m_data
220

FIG. 12D

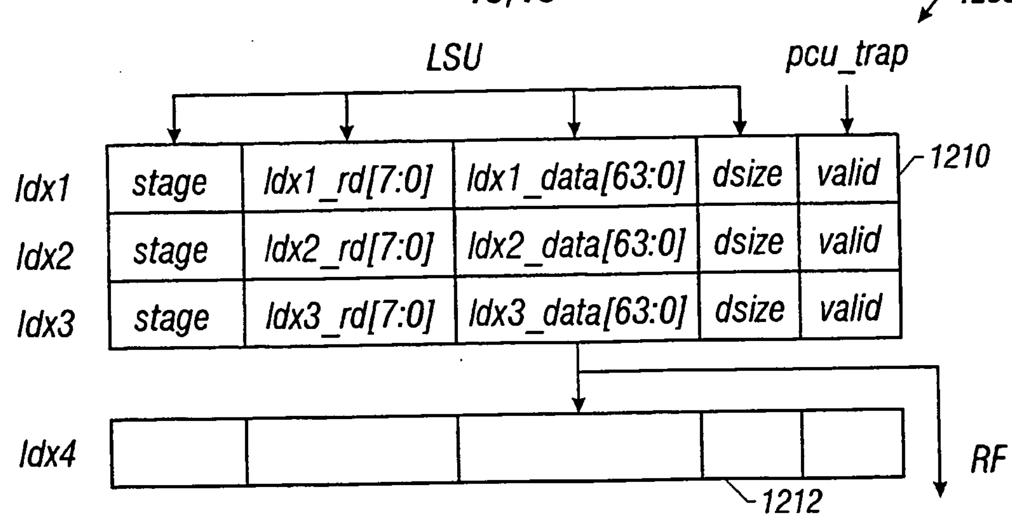



FIG. 13

| cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| load | D | E | C | A2 | A3 | T | WB | | | | | |
| membar | | D | E | E | E | A1 | A2 | A3 | T | WB | | |
| instrx | | | | | D | E | A1 | A2 | A3 | T | WB | |

FIG. 14A

| cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| setir | D | E | C | A2 | A3 | T | WB | | | | | |
| load | | D | E | C | A2 | A3 | T | WB | | | | |
| load | | | D | E | E | E | A1 | A2 | A3 | T | WB | |
| instrx | | | | | | D | E | A1 | A2 | A3 | T | WB |

FIG. 14B

# INTERNATIONAL SEARCH REPORT

International Application No
PCT/US 99/28822

**A. CLASSIFICATION OF SUBJECT MATTER**
IPC 7 G06F9/38

According to International Patent Classification (IPC) or to both national classification and IPC

**B. FIELDS SEARCHED**

Minimum documentation searched (classification system followed by classification symbols)
IPC 7 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

**C. DOCUMENTS CONSIDERED TO BE RELEVANT**

| Category ° | Citation of document, with indication, where appropriate, of the relevant passages | Relevant to claim No. |
|---|---|---|
| E | EP 0 962 856 A (TEXAS INSTRUMENTS INC) 8 December 1999 (1999-12-08) the whole document --- | 1-7,9, 11-16, 18-29, 32,33 |
| X | FILLO M ET AL: "THE M-MACHINE MULTICOMPUTER" PROCEEDINGS OF THE ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE,US,LOS ALAMITOS, IEEE COMP. SOC. PRESS, vol. SYMP. 28, 1995, pages 146-156, XP000585356 ISBN: 0-8186-7349-4 | 1-7,9, 11-16, 18-24, 26-29, 32,33 |
| Y | the whole document --- | 8,10 |
| | -/-- | |

[X] Further documents are listed in the continuation of box C.

[X] Patent family members are listed in annex.

° Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

| Date of the actual completion of the international search | Date of mailing of the international search report |
|---|---|
| 7 April 2000 | 17/04/2000 |
| **Name and mailing address of the ISA** | **Authorized officer** |
| European Patent Office, P.B. 5818 Patentlaan 2 NL – 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016 | Klocke, L |

1

# INTERNATIONAL SEARCH REPORT

Int. ..tional Application No
PCT/US 99/28822

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

| Category ° | Citation of document, with indication, where appropriate, of the relevant passages | Relevant to claim No. |
|---|---|---|
| X | WOLFE A ET AL: "A VARIABLE INSTRUCTION STREAM EXTENSION TO THE VLIW ARCHITECTURE" COMPUTER ARCHITECTURE NEWS, US, ASSOCIATION FOR COMPUTING MACHINERY, NEW YORK, vol. 19, no. 2, 1 April 1991 (1991-04-01), pages 2-14, XP000203245 ISSN: 0163-5964 the whole document --- | 1,3,4,6, 12,13, 20-22, 25,26, 28,29,32 |
| X | US 5 784 630 A (KOBAYASHI YOSHIKI ET AL) 21 July 1998 (1998-07-21) the whole document --- | 1,12,20, 26,28 |
| Y | GLOSSNER AND VASSILIADIS: "The DELFT-JAVA engine: an introduction" 3RD INTERNATIONAL EURO-PAR CONFERENCE , 26 - 29 August 1997, pages 29766-770, XP000901534 PASSAU, DE | 8,10 |
| A | the whole document --- | 17 |
| A | TULLSEN D M ET AL: "SIMULTANEOUS MULTITHREADING MAXIMIZING ON-CHIP PARALLELISM" PROCEEDINGS OF THE ANNUAL SYMPOSIUM ON COMPUTER ARCHITECTURE, US, NEW YORK, ACM, vol. SYMP. 22, 1995, pages 392-403, XP000687825 ISBN: 0-7803-3000-5 page 399, right-hand column -page 401, left-hand column --- | 1,12,20, 26,28,32 |
| A | MENDELSON AND MENDELSON: "Toward a general-purpose multi-stream system" PACT '94 : CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, 24 - 26 August 1994, pages 335-338, XP000571393 Montréal, CA the whole document ----- | 1,12,20, 26,28,32 |

1

# INTERNATIONAL SEARCH REPORT

Information on patent family members

International Application No
PCT/US 99/28822

| Patent document cited in search report | Publication date | Patent family member(s) | Publication date |
|---|---|---|---|
| EP 0962856 A | 08-12-1999 | JP 2000029731 A | 28-01-2000 |
| US 5784630 A | 21-07-1998 | JP 4117540 A | 17-04-1992 |
| | | JP 2834298 B | 09-12-1998 |
| | | JP 4127351 A | 28-04-1992 |
| | | US 5968160 A | 19-10-1999 |
| | | DE 4129614 A | 19-03-1992 |